Separation Logic Contracts for a Java-like Language with Fork/Join

Christian Haack, Clément Hurlin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

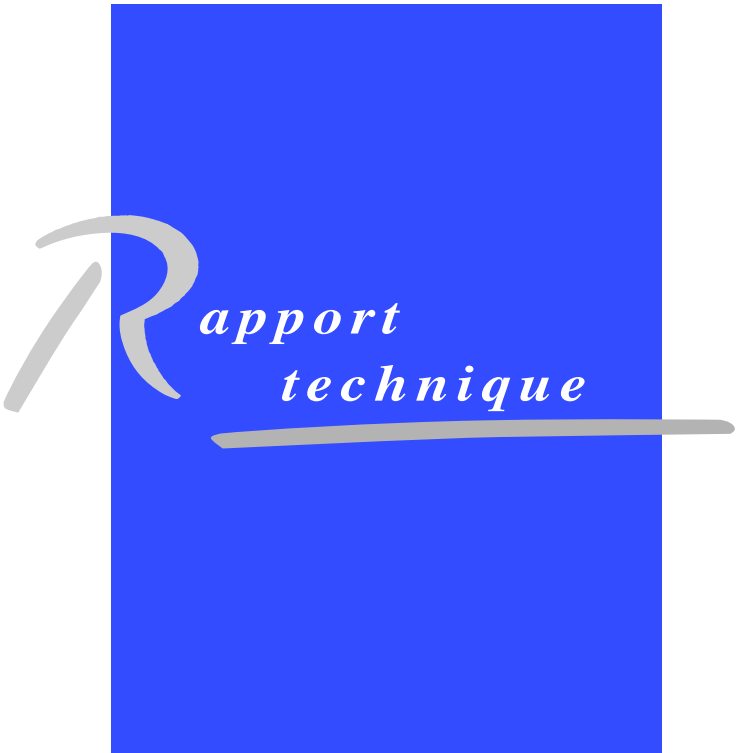
Separation Logic Contracts for a Java-like Language with Fork/Join

Christian Haack — Clément Hurlin

N° ????

Janvier 2008

Thème SYM



ISSN 0249-0803 ISRN INRIA/RT--????--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

Separation Logic Contracts for a Java-like Language with Fork/Join

Christian Haack* , Clément Hurlin*†

Thème SYM — Systèmes symboliques
Équipe-Projet Everest et Université Radboud de Nimègue

Rapport technique n° ???? — Janvier 2008 — 98 pages

Abstract: We adapt a variant of permission-accounting separation logic to a concurrent Java-like language with fork/join. To support both concurrent reads and information hiding, we combine fractional permissions with abstract predicates. We present a separation logic contract for iterators that prevents data races and concurrent modifications. Our program logic is presented in an algorithmic style, based on a proof-theoretical logical consequence. We show that verified programs satisfy the following properties: data race freedom, absence of null-dereferences and partial correctness.

Key-words: Program Verification, Separation Logic, Object-Orientation, Concurrency.

* Supported in part by IST-FET-2005-015905 Mobius project.
† Supported in part by ANR-06-SETIN-010 ParSec project.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Contrats en logique de séparation pour un langage à la Java avec fork/join

Résumé : Nous adaptons une variante de la logique de séparation avec permissions à un langage à la Java avec fork/join. Afin d'autoriser les lectures concurrentes dans le tas sans révéler l'implémentation, nous combinons les permissions fractionnelles avec les prédicats abstraits. Nous présentons une spécification d'itérateurs concurrents qui empêche les *data races* et les modifications concurrentes. Notre logique est présentée dans un style algorithmique, à partir d'un système de preuve théorique. Nous démontrons que les programmes vérifiés satisfont les propriétés suivantes: pas de *data race*, pas de déréference de pointeurs nuls et correction partielle.

Mots-clés : Vérification de programmes, logique de séparation, orienté objet, concurrence.

Contents

1 Introduction

As the trend to use multi-core hardware is growing, the need for verification of concurrent programs is becoming an important issue for mainstream programming. Programming with concurrency primitives, such as threads or processes, however, is made difficult by the possibilities of data races and deadlocks. To tackle these problems programmers often think in terms of *permissions*. To access a piece of memory a thread must have the permission to do so. As objects are created and passed between threads, permissions to access these objects are passed around. Ultimately, the use of a permission model eases verification of functional properties and prevents data races.

In separation logic, controlling access to memory space plays a prominent role. Separation logic formulas represent access tickets to heap space and programs must prove the possession of tickets before reading from or writing to memory. Separation logic contracts specify access policies. Adherence to these policies is checked statically by separation logic rules. Access policies are tightly coupled with assertions about memory content, so that it is impossible to maintain assertions that can be invalidated, for instance, by thread interference or memory updates through unknown aliases. Thus, separation logic perfectly supports reasoning about access permissions.

Separation logic has been invented in the beginning of this millennium [17, 29] as a formalism for reasoning about programs with aliasing and was originally applied to manually verify intricate pointer-manipulating algorithms (e.g., [32]). It was later realized that separation logic can be extended to reason about concurrent programs [24, 10, 6, 11]. Among other things, concurrent separation logic can be used to verify data race freedom (i.e., the absence of concurrent read/write or write/write accesses to the same memory location). Data race freedom is notoriously hard to verify. It is an important property, because data races result in unpredictable program behaviour [23, 30]. While the original work on separation logic was conceptual, more recently an experimental automatic checker for a subset of separation logic has been developed for programs written in a low-level model language [4]. Currently, the same group of researchers combines ideas from separation logic with automatic program analysis techniques in order to analyze systems code [14]. Separation logic has also been applied to an intermediate language for a C compiler with the goal of producing a fully verified compiler [2]. So far, only little work on applying separation logic to object-oriented programs exists [26, 27], and this work has only been theoretical. In addition, there are some recent object-oriented type and effect systems [9] and typestate systems [5], which are related to separation logic. Thus, while it seems to be apparent to a number of researchers that separation logic can be a very valuable tool for verifying properties of object-oriented programs, much works needs to be done to make this idea practical.

[Chris says: I got rid of the "clumsy" part. I think you complained about that earlier. I must have forgotten to delete it.]

In this paper, we present an adaptation of concurrent separation logic to a Java-like language with fork/join. We support concurrent reads by means of fractional permissions [8, 6]. Furthermore, we support object-oriented abstractions through abstract

predicates [27]. Abstract predicates can be understood as a generalization of type-states [13], object invariants [3] and data groups [22]. The combination of fractional permissions and abstract predicates proved a bit challenging. In order to split and merge access permissions for entire data groups, we had to treat data groups in a distinguished way. We combine fractional permissions with fork/join. As far as we know, until now fractional permissions have only been formalized for languages with a parallel composition operator, which is both cleaner and easier to formalize, but less realistic. We allow separation logic contracts to dereference `final` fields without any restrictions, just like stack variables. This often makes contracts more readable. To be able to verify abstract predicates in the presence of subclassing, we axiomatize the "stack of class frames" [13, 3] in separation logic. We present a separation logic specification of Java's `Iterator` that statically prevents concurrent modifications of the underlying collection. On the technical side, we present Hoare rules in an algorithmic style based on a proof-theoretical logical consequence judgment. This differs from most other presentations of separation logic, where logical consequence is defined model-theoretically. We prefer a proof-theoretical logical consequence, because that seems more amenable for algorithmic verification, which is our ultimate goal.

Section 2 informally introduces features of our system by example. The remainder of the paper is technical: Section 3 presents a Java-like model language, Section 4 the resource semantics of formulas, Section 5 their proof theory, Section 6 the verification rules, Section 7 the preservation theorem. Section 8 concludes.

2 Separation Logic Contracts for Java-like Programs

In this section, we show uses of separation logic contracts by example.

2.1 Separation Logic — Formulas as Access Tickets

Separation logic [17, 29] combines the usual logical operators with the points-to predicate $x.f \mapsto v$, the resource conjunction $F * G$ and the resource implication $F \mathrel{-\!\!*} G$.

The predicate $x.f \mapsto v$ has a *dual purpose*: firstly, it asserts that the object field $x.f$ contains data value v and, secondly, it represents a *ticket* that grants permission to access the field $x.f$. This is formalized by separation logic's Hoare rules for reading and writing fields:

$$\{x.f \mapsto \_ * F\}x.f = v\{x.f \mapsto v * F\} \qquad \{x.f \mapsto v * F\}y = x.f\{x.f \mapsto v * v == y * F\}$$

The crucial difference to standard Hoare logic is that both these rules have a precondition of the form $x.f \mapsto \_$[1]: this formula functions as an *access ticket* for $x.f$.

It is important that tickets are not forgeable. One ticket is not the same as two tickets! For this reason, the resource conjunction $*$ is not idempotent: F is not equivalent to $F * F$. The resource implication $\mathrel{-\!\!*}$ matches the resource conjunction $*$, in the sense that the modus ponens law is satisfied: $F * (F \mathrel{-\!\!*} G)$ implies G. However, $F * (F \mathrel{-\!\!*} G)$ *does not* imply $F * G$. In English, $F \mathrel{-\!\!*} G$ is pronounced as "consume F yielding G". In terms of tickets, $F \mathrel{-\!\!*} G$ permits to trade ticket F and receive ticket G in return.

Separation logic is particularly useful for concurrent programs: two concurrent threads simply split the resources that they may access, as formalized by the rule for the parallel composition $t \mid t'$ of threads t and t' [24].

[1] $x.f \mapsto \_$ is short for $(\exists v)(x.f \mapsto v)$.

$$\frac{\{F\}t\{G\} \quad \{F'\}t'\{G'\}}{\{F*F'\}t \mid t'\{G*G'\}}$$

With this concurrency rule, separation logic prevents data races. There is a caveat, though. The rule does not allow concurrent reads. Boyland [8] solved this problem with a very intuitive idea, which was later adapted to separation logic [6]. The idea is that (1) *access tickets are splittable*, (2) *a split of an access ticket still grants read access* and (3) *only a whole access ticket grants write access*. To account for multiple splits, Boyland uses fractions, hence the name *fractional permissions*. In permission-accounting separation logic [6], access tickets $x.f \mapsto v$ are superscripted by fractions π. $x.f \overset{\pi}{\longmapsto} v$ is equivalent to $x.f \overset{\pi/2}{\longmapsto} v * x.f \overset{\pi/2}{\longmapsto} v$. In the Hoare rules, writing requires the full fraction 1, whereas reading just requires *some* fraction π:

$$\{x.f \overset{1}{\longmapsto} \_ * F\}x.f = v\{x.f \overset{1}{\longmapsto} v * F\} \qquad \{x.f \overset{\pi}{\longmapsto} v * F\}y = x.f\{x.f \overset{\pi}{\longmapsto} v * v == y * F\}$$

Permission-accounting separation logic maintains the global invariant that the sum of all fractional permissions to the same cell is always at most 1. This prevents read-write and write-write conflicts, but permits concurrent reads.

In our Java-like language, we use ASCII and write `Perm(`$x.f,\pi$`)` for $x.f \overset{\pi}{\longmapsto} \_$, and `PointsTo(`$x.f,\pi,v$`)` for $x.f \overset{\pi}{\longmapsto} v$.

2.2 Separation Logic and Modifies Clauses

Object-oriented specification languages, like for instance JML [12], use `modifies` clauses to express frame conditions:

```
modifies this.f;
void set(int x) { this.f = x; }
```

Pre/postconditions in separation logic can be used to a similar effect:

```
req Perm(this.f,1); ens Perm(this.f,1);
void set(int x) { this.f = x; }
```

In separation logic, method preconditions specify what access permissions are required to execute the method body. Method postconditions specify what permission are passed back to the caller upon method return. Methods can loose permissions by forking new threads that require permissions. Methods can gain permissions by joining threads and picking up access permissions that the joined threads do not need anymore.

2.3 Separation Logic and Abstraction

Several object-oriented specification methodologies have abstraction features that allow exporting the name of an abstraction to object clients, while hiding its concrete definition. Examples include the *Inv* predicate in the Boogie methodology [3] (which indicates to object clients if the object invariant holds or not without exposing its definition), typestates for objects [13] and data groups [22]. Parkinson and Bierman [27] study abstractions of exactly this kind, where the (hidden) concrete definitions of the abstractions are given in terms of separation logic formulas. We build on their work, by adding support for concurrency and fractional permissions.

In our model language, interfaces may declare *abstract predicates* and classes may implement them by providing concrete definitions as separation logic formulas.

```
interface I { ... pred P<T̄ x̄>; ... group P<T̄ x̄>; ... }
class C implements I { ... pred P<T̄ x̄>=F; ... group P<T̄ x̄>=F; ... }
```

Like Parkinson/Bierman [27], but unlike the other examples mentioned above [3, 13, 22], we allow abstract predicates to have parameters in addition to the implicit self-parameter (as listed in the typed formal parameter lists $\bar{T}\,\bar{x}$). The types $\bar{T}$ for predicate parameters range over all Java types and the distinguished type `perm` for fractional permissions. Unlike Parkinson/Bierman, we differentiate between data groups and arbitrary predicates through the keywords `group` and `pred`. Both `group`s and `pred`s are defined by separation logic formulas, but the formulas that may define `group`s are restricted. As a result of this restriction, we obtain an equivalence between formulas $o.P\langle\pi\rangle$ and $o.P\langle\pi/2\rangle * o.P\langle\pi/2\rangle$ for `group`s P but not for `pred`s P. This equivalence is crucial for supporting concurrent reads in combination with abstract predicates.

We assume that the `Object` class declares a distinguished data group `state`:

```
class Object { group state<perm p> = true; }
```

The `state` data group represents the access permissions for the *object state*. The object state often consists exactly of the object's fields, but sometimes extends beyond the fields to include owned objects. Every class must extend the `state` data group and thereby define what the object states of its instances are. Our syntax for data group extensions (and similarly for predicate extensions) is as follows:

```
class C extends D { ... extends group P<T̄ x̄> by F;  ... }
```

Semantically the extension F of abstract predicate P gets $*$-*conjoined* with P's definition in C's superclass D. This is more restrictive than Parkinson/Bierman [27], who allow arbitrary predicate redefinitions in subclasses. On the upside, this restrictiveness enhances modularity by avoiding reverification of inherited methods, which is needed in Parkinson/Bierman's system.

We now borrow an example from Leino [22] to show how data groups can be specified in separation logic style.

```
class Sprite implements SpriteInt ext Object {
   protected int x,y;
   private int col;

   group position<perm p>   = Perm(x,p) * Perm(y,p);
   group color<perm p>      = Perm(col,p);
   extends group state<perm p> by position<p> * color<p>;

   req position<1>; ens position<1>;
   void updatePosition() { }

   req color<1>; ens color<1>;
   void updateColor() { }

   req state<1>; ens state<1>;
   void update() { updatePosition(); updateColor(); }

   req state<p>; ens state<p>;
   void display() {...}      // only read the state
}
```

Here, the `position` data group consists of the position fields `x` and `y`, the `color` data group of the field `c`. The `state` data group is their union. The three update-methods require write access to the corresponding data groups. Hence, their preconditions initialize the data group permission parameters by 1. The `display` method is readonly. Consequently, in its precondition the permission parameter of `state` can be an arbitrary fraction `p`. (Free variables in method contracts are implicitly universally quantified.)

The `Sprite` class implements the following interface:

```
interface SpriteInt {
   group position<perm p>;
   group color<perm p>;
   req position<1>; ens position<1>;
   void updatePosition();
   req color<1>; ens color<1>;
   void updateColor();
   req state<1>; ens state<1>;
   void update();
   req state<p>; ens state<p>;
   void display();
}
```

As is, this interface does not reveal that both `position` and `color` are subgroups of `state`. It is sometimes useful for object clients to know about this fact. Leino's language [22] can export facts about data group nesting to clients. In our language, this is facilitated by *class axioms*. Class axioms export facts about relations between abstract predicates, without revealing the detailed predicate implementations. Class implementors have to prove class axioms and class clients can use them. To export the fact that `position` and `color` are nested in `state` we add the following class axioms to the interface `SpriteInt` (which uses a typed universal quantifier `fa` over permissions `p`):

```
axiom (fa perm p)(position<p> ispartof state<p>);
axiom (fa perm p)(color<p> ispartof state<p>);
```

The formula "F `ispartof` G" is a derived form for G `-*` $(F$ `*` $(F$ `-*` $G))$. Intuitively, this formula says that F is a physical part of G: one can take G apart into F and its complement F `-*` G, and one can put the two parts back together to obtain G again.

2.4 Splitting and Merging Data Groups

We show how our system supports concurrent reads in combination with data group abstractions. To this end, we use an important law that we have shown sound. We call this law the *split/merge law* and paraphrase it as *splitting data group parameters splits data groups*. Specialized to the `state` data group the law looks like this (where F `*-*` G abbreviates $(F$ `-*` $G)$ `&` $(G$ `-*` $F)$):

$$o\texttt{.state<}p\texttt{>} \;\texttt{*-*}\; (o\texttt{.state<}p/2\texttt{>} \;\texttt{*}\; o\texttt{.state<}p/2\texttt{>})$$

In order to ensure that this law holds, we have to restrict the formulas that define data groups. For instance, the following definition would, quite obviously, break the law:

```
group state<perm p> = Perm(x.f,1);      disallowed!
```

The problem here is that `state`'s definition ignores the permission parameter, so that splitting the permission parameter leaves the data group permission intact. Data groups must be fully parameterized on their permissions. We also disallow occurrences of linear implications and disjunctions in data group definitions and only allow existentials with unique witnesses. Without these restrictions, our soundness proof for the split/merge law would not work.

To demonstrate a use of the split/merge law, consider the following example:

```
class Screen<perm p> extends Thread {
   final public Sprite sprite; // object to display
   req x.state<p>; ens sprite.state<p> * x==sprite;
   Screen(Sprite x) { this.sprite = x; }
   req sprite.state<p>; ens sprite.state<p>
   void run() { sprite.display(); }
}
```

We sketch a proof outline for a `Screen` client that forks two threads that concurrently display a sprite and then joins them again to gain the full permission on the sprite back. The example also illustrates how we deal with the concurrency primitives `fork`, `join` and `run`: Because `fork` calls `run`, we use `run`'s precondition as the precondition of `fork`. Because `join` waits until the `run` method has terminated, we use `run`'s postcondition as the postcondition of `join`. In order to avoid that several threads that all call x.`join()` on the same receiver simultaneously use the postcondition of x.`run()`, `Thread` constructors "return" a single ticket `Perm(`x`[join],1)` which is required when x.`join()`'s postcondition is used. Our proof rules even allow to split this `join`-ticket and split `join`'s postcondition correspondingly. This is useful in situations where several clients `join` the same thread x and then want to read-access x's state. In order to facilitate this kind of "read-only multi-joining", we impose restrictions on `run`'s postconditions, similar to the restrictions on data group definitions.

```
  { s.state<1> }
  (split/merge law from left to right)
  { s.state<1/2> * s.state<1/2> }
Screen<1/2> scr1 = new Screen<1/2>(s);
Screen<1/2> scr2 = new Screen<1/2>(s);
  { scr1.sprite.state<1/2> * s==scr1.sprite * Perm(scr1[join],1) *
    scr2.sprite.state<1/2> * s==scr2.sprite * Perm(scr2[join],1) }
scr1.fork();
  { s==scr1.sprite * Perm(scr1[join],1) *
    scr2.sprite.state<1/2> * s==scr2.sprite * Perm(scr2[join],1) }
scr2.fork();
  { s==scr1.sprite * Perm(scr1[join],1) *
    s==scr2.sprite * Perm(scr2[join],1) }
scr1.join();
  { s==scr1.sprite * scr1.sprite.state<1/2> *
    s==scr2.sprite * Perm(scr2[join],1) }
scr2.join();
```

```
{ s==scr1.sprite * scr1.sprite.state<1/2> *
  s==scr2.sprite * scr2.sprite.state<1/2> }
(substitutivity)
{ s.state<1/2> * s.state<1/2> }
(split/merge law from right to left)
{ s.state<1> }
```

Note that in this proof outline and in the postcondition of the `Screen` constructor, we dereference the `final` fields `scr1.sprite`, `scr2.sprite` and `this.sprite`. Usually, in separation logic dereferencing is only allowed in the leftmost argument of `PointsTo`. We support unrestricted dereferencing of `final` fields: this makes contracts more readable and `final` fields are common in Java programs.

2.5 Object Usage Protocols

Often one wants to constrain object clients to adhere to certain usage protocols. Usage protocols can, for instance, be specified in typestate systems [13] or, using ghost fields, by general purpose specification languages [28]. A limitation of these techniques is that state transitions must always be associated with method calls. This is sometimes not sufficient. Consider for instance a variant of Java's `Iterator` interface (enriched with an `init` method to avoid constructor contracts).

```
interface Iterator {
  void init(Collection c);
  boolean hasNext();
  Object next();
  void remove();
}
```

If iterators are used in an undisciplined way, there is the danger of unwanted concurrent modification of the underlying collection (both of the collection elements and the collection itself). Moreover, in concurrent programs bad iterator usage can result in data races. It is therefore important that `Iterator` clients adhere to a usage discipline. The following simple discipline would be safe for an iterator without `remove`: *retrieve the next collection element; then access the element; then trade the element access right for the right to retrieve the next element; and so on*. Although such a discipline is simple and makes sense, it cannot be specified by existing typestate systems and it would be very clumsy to specify it with classical specification languages.

We have designed a usage protocol for the full `Iterator` interface with `remove`. Its state machine is shown in Figure 1. The dashed arrows are the ones that are not associated with method calls, and are hard to capture with existing object-oriented specification systems. Note in particular, that according to this protocol an `Iterator` client can keep the access right for a collection element that he has removed. This protocol can be expressed quite straightforwardly by a separation logic contract (making heavy use of linear implication).

```
interface Iterator<perm p, Collection iteratee> {
   pred ready;                          // prestate for iteration cycle
   pred readyForNext;                   // prestate for next()
```

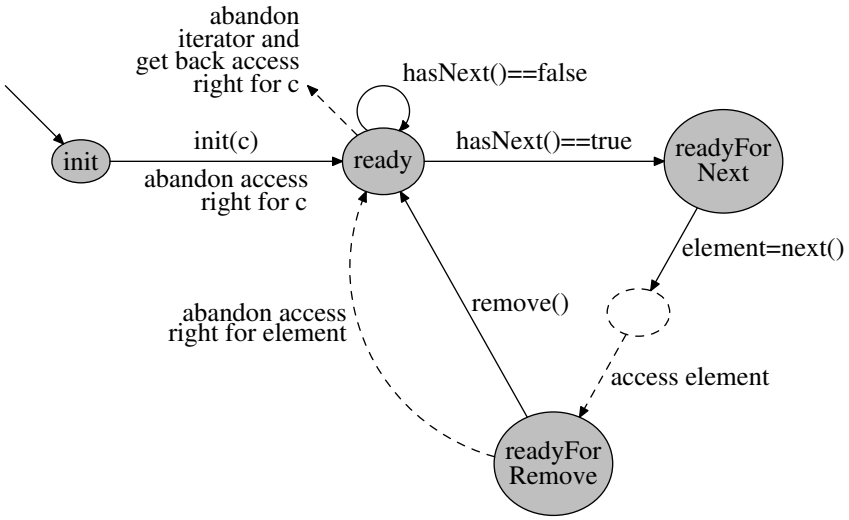


Figure 1: Usage Protocol of the `Iterator` interface

```
   pred readyForRemove<Object element>;  // prestate for remove()
   axiom ready -* iteratee.state<p>;     // stop iterating
   req init * c.state<p> * c==iteratee;
   ens ready;
   void init(Collection c);
   req ready;
   ens (result -* readyForNext) & (!result -* ready);
   boolean hasNext();
   req readyForNext;
   ens result.state<p> * readyForRemove<result> *
         ((result.state<p> * readyForRemove<result>) -* ready);
   Object next();
   req readyForRemove<_> * p==1;
   ens ready;
   void remove();
}
```

The interface has two parameters: firstly, a permission `p` and, secondly, the iteratee. If the permission parameter is instantiated by a fraction $p \leq 1$, one obtains a read-only iterator, otherwise a read-write iterator. The states are represented by three abstract predicates. The class axiom expresses that whenever the client is in the `ready` state, he has the option to abandon the iterator for good, and get the access right for the iteratee back. The precondition of `init()` consumes a fraction `p` of the access right for the iteratee and puts the iterator in the `ready` state. The `init` predicate in `init()`'s precondition is a special abstract predicate that every object enters right after object

creation and that grants access to all of the object's fields.[2] The most interesting part of the `Iterator` contract is the postcondition of `next()`. It grants access to the collection element that got returned, represented by the special `result` variable. Furthermore, it grants permission to remove this element. However, by the precondition of `remove`, this permission can only be used if the class parameter `p` is 1, i.e., the iterator is read-write. Finally, `next()`'s postcondition grants right to trade the tickets `result.state<p>` and `readyForRemove<result>` for the `ready` state.

We have implemented this interface for a doubly linked list implementation of the `Collection` interface (see Appendix A.3).

3 The Model Language

We use the same syntax conventions as Featherweight Java (FJ) [16]. In particular, we indicate sequences of X's by an overbar: $\bar{X}$. We sometimes write $\bar{X}_{\mathsf{to}\,n}$ for $\bar{X}$'s prefix of length n. We also use regular expression notation: X**?** for an optional X, X***** for a possibly empty list of X's, $X \mid Y$ for an X or a Y, and $X\ Y$ for X followed by Y. For any syntactic object X, we let $\mathsf{fv}(X)$ be the set of free variables of X. We often write $x \notin X$ to abbreviate $x \notin \mathsf{fv}(X)$.

Identifier Domains:

$C, D \in \mathsf{ClassId}$	class identifiers (including `Object`)
$I, J \in \mathsf{IntId}$	interface identifiers
$s, t \in \mathsf{TyId} = \mathsf{ClassId} \cup \mathsf{IntId}$	type identifiers
$o, p, q \in \mathsf{ObjId}$	object identifiers
$f \in \mathsf{FieldId}$	field identifiers
$m, n \in \mathsf{MethId}$	method identifiers
$P^{\mathsf{grp}} \in \mathsf{GrpId}$	data group identifiers (including `state`)
$P \in \mathsf{PredId} \supseteq \mathsf{GrpId}$	predicate identifiers (including `init`)
$\iota \in \mathsf{RdVar}$	read-only variables (including `this`)
$\ell \in \mathsf{RdWrVar}$	read-write variables
$\alpha^{\mathsf{perm}} \in \mathsf{PermVar}$	logic variables for permissions
$\alpha^{\mathsf{val}} \in \mathsf{LogValVar}$	logic variables for values (including `result`)
$\alpha \in \mathsf{LogVar} = \mathsf{PermVar} \cup \mathsf{LogValVar}$	logic variables
$x, y, z \in \mathsf{Var} = \mathsf{RdVar} \cup \mathsf{RdWrVar} \cup \mathsf{LogVar}$	variables

We distinguish between read-only variables ι, read-write variables ℓ and logic variables α. Method parameters (including `this`) are read-only. Logic variables can only occur in specifications and types. They range over both fractional permissions and values (like integers, object identifiers and `null`). The special variable `result` is used in method postconditions to refer to the return value.

$n \in \mathsf{Int}$	integers	$b \in \mathsf{Bool} = \{\texttt{true}, \texttt{false}\}$		booleans
$u, v, w \in \mathsf{Val}$	::=	`null` $\mid n \mid b \mid o \mid \iota$		values
$\pi^{\mathsf{val}} \in \mathsf{LogVal}$	::=	$\alpha^{\mathsf{val}} \mid v$		logic values
$\pi^{\mathsf{perm}} \in \mathsf{Perm}$	::=	$\alpha^{\mathsf{perm}} \mid 1 \mid \texttt{split}(\pi^{\mathsf{perm}})$		permissions
$\pi \in \mathsf{SpecVal}$	::=	$\pi^{\mathsf{val}} \mid \pi^{\mathsf{perm}}$		specification values
$T, U, V, W \in \mathsf{Ty}$	::=	`void` $\mid$ `int` $\mid$ `bool` $\mid t\texttt{<}\bar{\pi}\texttt{>} \mid$ `perm`		types

[2] Our model language does not have constructors.

We include read-only variables (but not read-write variables) in the syntax domain of *values*. This is convenient for our substitution-based operational semantics. *Fractional permissions* are represented symbolically: $\texttt{split}^n(1)$ represents the concrete fraction $\frac{1}{2^n}$. In examples, we sometimes write $\frac{1}{2^n}$ as syntax sugar for $\texttt{split}^n(1)$. *Specification values* union logic values and permissions. For later convenience, we extend the syntactic `split`-operation to specification values: $\mathsf{split}(\pi^{\mathsf{perm}}) = \texttt{split}(\pi^{\mathsf{perm}})$ and $\mathsf{split}(\pi^{\mathsf{val}}) = \pi^{\mathsf{val}}$. Interfaces and classes are parameterized by specification values. Correspondingly, object types $t\texttt{<}\overline{\pi}\texttt{>}$ instantiate the parameters. We usually omit the angle brackets, if the parameter list is empty.

Interface Declarations:

$F \in \mathsf{Formula} ::= \ldots$	specification formulas (defined in Section 4)
$spec ::= \texttt{req}\ F\texttt{;}\ \texttt{ens}\ F\texttt{;}$	pre- and postconditions
$mt ::= \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\overline{V}\ \overline{\imath}\texttt{)}$	method types (scope of $\overline{\alpha},\overline{\imath}$ is $\overline{T}, spec, U, \overline{V}$)
$pmod ::= \texttt{pred} \mid \texttt{group}$	predicate modifiers
$pt ::= pmod\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}$	predicate types
$ax ::= \texttt{axiom}\ F$	class axioms
$int \in \mathsf{Interface} ::= \texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ \overline{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*\}}$	interfaces (scope of $\overline{\alpha}$ is $\overline{T}, \overline{U}, pt\texttt{*}, ax\texttt{*}, mt\texttt{*}$)

Syntactic restriction: The type "`perm`" may only occur inside angle brackets or formulas.

Method types include pre- and postconditions and are parameterized by logic variables. In examples, we often leave these quantifiers over logic variables implicit, but prefer to treat them explicitly in the formal language. *Class axioms* can export useful facts about predicate implementations to class clients, without exposing predicate implementations in detail. In class axioms, we often omit leading universal quantifiers, if the type of these variables can be inferred from the context.

Class Declarations:

$fin ::= \texttt{final?}$	optional final modifier
$fd ::= T\ f$	field declarations
$pd ::=$	predicate definitions
$\quad fin\ pmod\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>=}F$	root definition (scope of $\overline{\alpha}$ is F)
$\quad fin\ \texttt{ext}\ pmod\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{by}\ F$	extension (scope of $\overline{\alpha}$ is F)
$md ::= fin\ \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ spec\ U\ m\texttt{(}\overline{V}\ \overline{\imath}\texttt{)\{}c\texttt{\}}$	method (scope of $\overline{\alpha},\overline{\imath}$ is $\overline{T}, spec, U, \overline{V}, c$)
$cl \in \mathsf{Class} ::= fin\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \overline{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*\}}$	class (scope of $\overline{\alpha}$ is $\overline{T}, U, \overline{V}, fd\texttt{*}, pd\texttt{*}, ax\texttt{*}, md\texttt{*}$)
$ct \subseteq \mathsf{Interface} \cup \mathsf{Class}$	class tables

Syntactic restrictions:

- The type "`perm`" may only occur inside angle brackets or specification formulas.
- Cyclic predicate definitions in *ct* must be positive.

The *first syntactic restriction* ensures that fractional permissions do not spill into the executable part of the language. The *second syntactic restriction* ensures that predicate implementations (which can be recursive) are well-founded. This restriction is more liberal than Parkinson/Bierman's restriction [27], who entirely prohibit predicate occurrences in negative positions (i.e., to the left of an odd number of implications) in

predicate implementations. We allow negative dependencies of predicate P on predicate Q as long as Q does not also depend on P. We need this additional freedom in the implementation of our `Iterator` interface

We use the symbol $\preceq_{ct}$ for the order on type identifiers induced by class table *ct*. We often leave the subscript *ct* implicit. We impose the following sanity conditions on *ct*: (1) $\preceq_{ct}$ is antisymmetric, (2) if t (except `Object`) occurs anywhere in *ct* then t is declared in *ct* and (3) *ct* does not contain duplicate declarations or a declaration of `Object`. We write $\mathsf{dom}(ct)$ for the set of all type identifiers declared in *ct*.

Subtyping is inductively defined by the following rules:

$$T <: T \qquad T <: U, U <: V \Rightarrow T <: V \qquad s\texttt{<}\overline{T}\,\overline{\alpha}\texttt{>}\ \texttt{ext}\ t\texttt{<}\overline{\pi'}\texttt{>} \Rightarrow s\texttt{<}\overline{\pi}\texttt{>} <: t\texttt{<}\overline{\pi'}[\overline{\pi}/\overline{\alpha}]\texttt{>}$$
$$t\texttt{<}\overline{\pi}\texttt{>} <: \texttt{Object} \qquad t\texttt{<}\overline{T}\,\overline{\alpha}\texttt{>}\ \texttt{impl}\ I\texttt{<}\pi'\texttt{>} \Rightarrow t\texttt{<}\overline{\pi}\texttt{>} <: I\texttt{<}\pi'[\overline{\pi}/\overline{\alpha}]\texttt{>}$$

We assume that class tables always contain the following class declaration:

```
class Thread ext Object {
  final void fork(); final void join();
  req false; ens true; void run() { null }
}
```

The methods `fork` and `join` do not have implementations. Instead, the operational semantics treats them in a special way:

- *o*`.fork()` creates a new thread, whose thread identifier is *o*, and executes *o*`.run()` in this thread. The *o*`.fork`-method should not be called more than once (on the same receiver *o*). A second call results in blocking.
- *o*`.join()` blocks until thread *o* has terminated.

The `run`-method is meant to be overridden. The pre/postconditions for `Thread.run()` are chosen so that they do not impose any restrictions on overriding this method. The pre/postconditions for `fork` and `join` are omitted, because our verification system ignores them anyways. Instead, it uses the precondition for `run` as the precondition for `fork` and the postcondition for `run` as the postcondition for `join`.

Commands:

$op \in \mathsf{Op} \supseteq \{\texttt{==}, \texttt{!}, \texttt{\&}, \texttt{|}\} \cup \{C\ \texttt{isclassof} \mid C \in \mathsf{ClassId}\}$

$c \in \mathsf{Cmd} ::=$	commands
v	return value (or `null` in case of type `void`)
$T\ \ell; c$	local variable declaration (scope of ℓ is c)
`final` $T\ \iota = \ell; c$	local read-only variable declaration (scope of ι is c)
`unpack (ex` $T\ \alpha$`)(`F`);` c	unpacking an existential (scope of α is F, c)
$hc; c$	first do hc, then do c

$hc \in \mathsf{HeadCmd} ::= \ell = v \mid \ell = op(\overline{v}) \mid \ell = v.f \mid \mathit{fin}\ v.f = v \mid \ell = \texttt{(}T\texttt{)}v \mid \ell = \texttt{new}\ C\texttt{<}\overline{\pi}\texttt{>} \mid$
$\texttt{if (}v\texttt{)\{}c\texttt{\}else\{}c'\texttt{\}} \mid \ell = v.m\texttt{<}\overline{\pi}\texttt{>(}\overline{v}\texttt{)} \mid \texttt{assert(}F\texttt{)}$

Synt. Restr.: Logic variables that occur in $\ell = \texttt{new}\ C\texttt{<}\overline{\pi}\texttt{>}$ must be bound by class parameters.

For brevity, we leave the `return`-*command implicit*. Values are included in the syntax domain of commands, so that a terminating, non-blocking execution of a command results in the return value. Methods of type `void` return `null`, which is the only member of type `void`. We usually omit terminating occurrences of `null`.

In *local variable declarations*, we treat ℓ and $\imath$ as binders with scope c. We identify commands up to renaming of bound variables, provided the renaming maps read-only to read-only and read-write to read-write variables.

The operator for *existential unpacking* has no effect at runtime. It makes the existential variable α available in the continuation c for instantiation of logic method parameters. In examples, we often omit explicit existential unpacking and instantiation of logic method parameters. Making these explicit helps with the theory, but ideally in practice an algorithmic static checker would infer these.

Our language has *no composite expressions* (e.g., $x.f.g.m()$), but instead requires that intermediate results are always assigned to local variables. Furthermore, all variables that occur on right-hand sides of assignments are read-only. These syntactic requirements simplify the verification rules, but are not true restrictions, because programs without these restrictions can be translated to our core language by inserting assignments to local variables.

Field assignments can optionally be preceded by a `final` modifier to indicate that this is the last assignment to the field. Afterwards, the field is `final`, that is, permanently read-only. Assignments to `final` fields are forbidden and this policy is statically enforced by our verification system. Our `final` fields generalize Java's `final` fields. The difference is that Java's `final` fields have to be read-only right after object construction, whereas, in our system, fields can be finalized at an arbitrary execution point. This can be particularly useful for arrays and matrices, because Java provides no way to declare their fields `final`.

There is one (and only one) rule, where our operational semantics depends on class parameters, namely in the reduction rule for *type casts*. Downcasts to parameterized types require a runtime check that looks at the type parameters, which the standard JVM does not keep track of. There are at least three ways how one could deal with that in practice: Firstly (and most pragmatically), one could simply forbid downcasts to reference types that have a non-empty parameter list. (We did not use any downcasts to parameterized types in our examples.) Then our type cast operator would degenerate to Java's standard cast operator. Secondly, one could develop an enhanced virtual machine that keeps track of class parameters. Although this solution is both clean and general, the drawback is that it restricts verified code to run on such enhanced JVMs (if we want soundness). Thirdly, one could devise a sound syntactic translation that erases class parameters such that the target of this translation throws a `ClassCastException` whenever the source of the translation does. A possible translation would encode class parameters as ghost fields and translate type casts of the form "$\ell = (C\texttt{<}\bar{\pi}\texttt{>})\,v$" to class casts "$\ell = (C)\,v$" followed by a sequence of tests that compare ℓ's ghost fields with the class parameters $\bar{\pi}$ and trigger a class cast exception if one of these tests fails.

Runtime Structures:

$\mathsf{ClVal} = \mathsf{Val} \setminus \mathsf{RdVar}$	closed values
$s \in \mathsf{Stack} = \mathsf{RdWrVar} \rightharpoonup \mathsf{ClVal}$	stacks
$t \in \mathsf{Thread} = \mathsf{Stack} \times \mathsf{Cmd} \ ::= \ s \,\mathsf{in}\, c$	threads
$ts \in \mathsf{ThreadPool} = \mathsf{ObjId} \rightharpoonup \mathsf{Thread} \ ::= \ o_1 \,\mathsf{is}\, t_1 \mid \cdots \mid o_n \,\mathsf{is}\, t_n$	thread pools
$os \in \mathsf{ObjStore} = \mathsf{FieldId} \rightharpoonup \mathsf{ClVal}$	object stores
$obj \in \mathsf{Obj} = \mathsf{Ty} \times \mathsf{ObjStore} \ ::= \ (T, os)$	objects
$h \in \mathsf{Heap} = \mathsf{ObjId} \rightharpoonup \mathsf{Obj}$	heaps

$st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{ThreadPool}$	$::= \langle h, ts \rangle$	states
$prog \in \mathsf{Program} = \mathsf{ClassTable} \times \mathsf{Cmd}$	$::= (ct, c)$	programs

Each thread "s in c" consists of a thread-local stack s and a process continuation c. In thread pools, each thread t is associated with a unique object identifier, which serves as a thread identifier. The dynamic semantics of our language is a small-step operational semantics $st \rightarrow_{ct} st'$ and can be found in Appendix E. We map commands to *initial states*: $\mathsf{init}(c) = \langle \{\mathsf{main} \mapsto (\texttt{Thread}, \emptyset)\}, \mathsf{main\ is}\ (\emptyset\ \mathsf{in}\ c) \rangle$, where main is some distinguished object id for the main thread. The main thread has an empty set of fields (hence the first $\emptyset$), and its stack is initially empty (hence the second $\emptyset$).

Below, we define a verification system whose top level judgment is $prog : \diamond$ (read: "*prog* is verified"). We have proven a *preservation theorem* from which we can draw several corollaries, namely, *data race freedom*, *null error freedom* and a variant of *partial correctness*. With the model that we have developed up to here, we can state the first two corollaries; for partial correctness, we have to wait until we have presented specification formulas and their semantics.

A pair (hc, hc') of head commands is called a *data race* iff $hc = (fin\ o.f = v)$ and either $hc' = (fin'\ o.f = v')$ or $hc' = (\ell = o.f)$ for some $o, f, v, v', \ell, fin, fin'$.

Theorem 1 (Verified Programs are Data Race Free)
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, ts \mid o_1\ \mathsf{is}\ (s_1\ \mathsf{in}\ hc_1; c_1) \mid o_2\ \mathsf{is}\ (s_2\ \mathsf{in}\ hc_2; c_2) \rangle$, *then* (hc_1, hc_2) *is not a data race.*

A head command hc is called a *null error* iff $hc = (\ell = \texttt{null}.f)$ or $hc = (fin\ \texttt{null}.f = v)$ or $hc = (\ell = \texttt{null}.m\texttt{<}\overline{\pi}\texttt{>}(\overline{v}))$ for some $\ell, fin, f, v, m, \overline{\pi}, \overline{v}$.

Theorem 2 (Verified Programs are Null Error Free)
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ hc; c) \rangle$, *then hc is not a null error.*

4 Specification Formulas and Their Semantics

Specification formulas contain expressions:

$$e \in \mathsf{Exp} \quad ::= \quad \pi \mid \ell \mid op(\overline{e}) \mid e.f \qquad \text{expressions}$$

Unlike in standard separation logic, we allow expressions to contain field references. Our verification rules ensure that expressions in program derivations *only refer to* `final` *fields*. Although our special treatment of `final` fields does not increase the expressivity of separation logic (because in separation logic one can follow reference chains into the heap by combining the points-to predicate with existential quantification), directly referring to values that `final` fields point to, often makes specifications more readable.

Specification Formulas:

$lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}, \texttt{\|}\}$	logical operators	$qt \in \{\texttt{ex}, \texttt{fa}\}$	quantifiers
$\kappa \in \mathsf{Pred} ::=$	predicates		
P	P at receiver's dynamic class		
$P\texttt{@}C$	P at class C		
$E, F, G, H \in \mathsf{Formula} ::=$	specification formulas		
e	boolean expression		

`PointsTo(`e`[`f`],`π`,`e'`)`	$e.f$ points to e' and the access permission for $e.f$ is π
`Perm(`e`[join],`π`)`	permission to use a split of `join`'s postcondition
`Pure(`e`)`	e is invariant under heap updates and evaluates normally
π`.`κ`<`$\overline{\pi'}$`>`	predicate $\pi.\kappa$ applied to $\overline{\pi'}$
F *lop* G	binary logical operator
`(`qt T α`)(`F`)`	quantifier

Syntactic restriction: In `(`qt T α`)(`F`)`, α does not occur inside field selection expressions $e.f$.

In concrete syntax, we often write `PointsTo(`$e.f$`,`π`,`e'`)` for `PointsTo(`e`[`f`],`π`,`e'`)`. (We choose square brackets in abstract syntax, because the syntactic restriction for quantifiers is easier to state that way.) The following derived forms are useful, too:

$$F \mathtt{\ *-*\ } G \triangleq (F \mathtt{\ -*\ } G) \mathtt{\ \&\ } (G \mathtt{\ -*\ } F) \qquad F \mathtt{\ assures\ } G \triangleq F \mathtt{\ -*\ } (F * G)$$
$$F \mathtt{\ ispartof\ } G \triangleq G \mathtt{\ -*\ } (F * (F \mathtt{\ -*\ } G))$$
$$\mathtt{Perm(}e\mathtt{[}f\mathtt{]},\pi\mathtt{)} \triangleq \mathtt{(ex}\ T\ \alpha\mathtt{)(\ PointsTo(}e\mathtt{[}f\mathtt{]},\pi,\alpha\mathtt{))} \quad \text{where } T \text{ is } e.f\text{'s least type}$$
$$e.\kappa\mathtt{<}\overline{\pi}\mathtt{>} \triangleq \mathtt{(ex}\ T\ \alpha\mathtt{)(}\alpha \mathtt{\ ==\ } e * \alpha.\kappa\mathtt{<}\overline{\pi}\mathtt{>)} \quad \text{where } T \text{ is } e\text{'s least type}$$

4.1 Resources

For the interpretation of resource conjunction, we define a heap joining operator. We want to allow splitting heaps on a per-field basis. To this end, it is convenient to define a function that maps heaps to functional relations. A *functional relation* is a downward closed subset $\overline{h}$ of $\mathsf{ObjId} \times \mathsf{Ty} \times (\mathsf{FieldId} \times \mathsf{ClVal})_\perp$ such that $(o,T,\perp),(o,T',\perp) \in \overline{h}$ implies $T = T'$ and $(o,T,(f,v)),(o,T,(f,v')) \in \overline{h}$ implies $v = v'$. Let FunRel be the set of all functional relations. We define the following bijections:

$$\backslash\backslash : \mathsf{Heap} \to \mathsf{FunRel} \quad \backslash\backslash h \triangleq \{\ (o,T,x) \mid h(o)_1 = T \ \wedge\ x \in h(o)_2 \cup \{\perp\}\ \}$$
$$/\!/ : \mathsf{FunRel} \to \mathsf{Heap} \quad /\!/\overline{h}(o)_1 \triangleq T, \text{ if } (o,T,\perp) \in \overline{h}\ \ /\!/\overline{h}(o)_2 \triangleq \{(f,v) \mid (o,/\!/\overline{h}(o)_1,(f,v)) \in \overline{h}\}$$

Now, we can define a partial operator $*$ that joins heaps and a heap order as follows:

$$\# \triangleq \{\ (h,h') \mid \backslash\backslash h \cup \backslash\backslash h' \in \mathsf{FunRel}\ \} \qquad * : \# \to \mathsf{Heap} \qquad h{*}h' \triangleq /\!/(\backslash\backslash h \cup \backslash\backslash h')$$
$$h \leq h' \quad \text{iff} \quad \backslash\backslash h \subseteq \backslash\backslash h'$$

We note that $*$ is commutative, associative and monotone with respect to $\leq$, that $h \leq h'$ iff $h' = h * h''$ for some h'', that arbitrary greatest lower bounds exist, $\bigwedge_{i\in I} h_i = /\!/ \bigcap_{i\in I} \backslash\backslash h_i$, and that least upper bounds of bounded sets exist, $\bigvee_{i\in I} h_i = /\!/ \bigcup_{i\in I} \backslash\backslash h_i$.

Our semantic domains for fractional permissions are *permission tables*. Let $[0,1]$ be the set of real numbers between 0 and 1: $[0,1] = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$.

Definition 1 (Permission Tables) A *permission table* is a total function of type $\mathsf{ObjId} \times (\mathsf{FieldId} \times \{\mathtt{join}\}) \to [0,1]$. Let $\mathsf{PermTable}$ be the set of all permission tables. Let meta-variables $\mathscr{P}, \mathscr{Q}$ range over permission tables.

Addition and subtraction on permission tables are defined pointwise. Obviously, these operations are partial, because $\mathscr{P} + \mathscr{Q}$ does not necessarily map into $[0,1]$, and similarly $\mathscr{P} - \mathscr{Q}$. We write $\mathscr{P}\#\mathscr{Q}$ whenever $\mathscr{P} + \mathscr{Q} \in \mathsf{PermTable}$. Division by 2 is also defined pointwise and is obviously total: if $\mathscr{P}$ maps into $[0,1]$, then so does $\frac{1}{2}\mathscr{P}$. $\mathbf{0}$ is the constant zero-function on $\mathsf{ObjId} \times (\mathsf{FieldId} \times \{\mathtt{join}\})$ and $\mathbf{1}$ the constant one-function. The order $\leq$ on permission tables is defined pointwise, $\mathscr{P} \wedge \mathscr{Q}$ and $\bigwedge_{i\in I} \mathscr{P}_i$

are the greatest lower bounds with respect to $\leq$, and $\mathscr{P} \vee \mathscr{Q}$ and $\bigvee_{i\in I} \mathscr{P}_i$ the least upper bounds.

In our model, resources are triples $(h, \mathscr{P}, \mathscr{Q})$ of a heap and two permission tables. Intuitively, $\mathscr{P}$ is a *local permission table* for a single thread: it records the thread's access permissions to heap cells in h. The *global permission table* $\mathscr{Q}$ is an upper bound on the sum of all permission tables of all threads. If a cell gets `finalized`, its entry in $\mathscr{Q}$ drops below 1 and the cell can be shared freely from that point on.

We define: A triple $(h, \mathscr{P}, \mathscr{Q}) \in \mathsf{Heap} \times \mathsf{PermTable} \times \mathsf{PermTable}$ is *sound* whenever the following conditions hold:

(a) $\mathsf{fst} \circ h \vdash h : \diamond$

(b) $\mathscr{P} \leq \mathscr{Q}$.

(c) For all $o \in \mathsf{dom}(h)$ and $f \in \mathsf{dom}(h(o)_2)$, either $\mathscr{P}(o,f) > 0$ or $\mathscr{Q}(o,f) < 1$.

(d) For all $o \notin \mathsf{dom}(h)$ and all k in $\mathsf{FieldId} \cup \{\texttt{join}\}$, $\mathscr{P}(o,k) = 0$ and $\mathscr{Q}(o,k) = 1$.

(e) For all o, f, if $\mathscr{Q}(o,f) < 1$ then $o \in \mathsf{dom}(h)$ and $f \in \mathsf{dom}(h(o)_2)$.

Condition (a) says that h must be well-typed. Condition (b) is a sanity condition that ensures that our intuitive interpretations of $\mathscr{P}$ and $\mathscr{Q}$ as local and global permission tables make sense. Condition (c) ensures that the partial heap h only contains cells that are either `final` or associated with a positive permission. Technically, this condition is needed to prove soundness of the verification rule for field updates. Condition (d) ensures that all objects that are not yet allocated have minimal permissions (with respect to the resource order presented below). This is needed to prove soundness of the verification rule for allocating new objects. Condition (e) ensures that all `final` heap cells are part of the heap. It is needed to prove substitutivity for pure expressions.

We define $\mathsf{Resources} = \{\ (h, \mathscr{P}, \mathscr{Q}) \mid (h, \mathscr{P}, \mathscr{Q}) \text{ is sound } \}$ and let meta-variable $\mathscr{R}$ range over Resources. For $\mathscr{R} = (h, \mathscr{P}, \mathscr{Q})$, let $\mathscr{R}_{\mathsf{hp}} = h$, $\mathscr{R}_{\mathsf{loc}} = \mathscr{P}$ and $\mathscr{R}_{\mathsf{glo}} = \mathscr{Q}$. Now we can define resource joining:

$$(h, \mathscr{P}, \mathscr{Q}) \# (h', \mathscr{P}', \mathscr{Q}') \quad \text{iff} \quad h \# h',\ \mathscr{P} \# \mathscr{P}',\ \mathscr{Q} = \mathscr{Q}' \text{ and } (h * h', \mathscr{P} + \mathscr{P}', \mathscr{Q}) \text{ is sound}$$
$$* : \# \to \mathsf{Resources} \qquad (h, \mathscr{P}, \mathscr{Q}) * (h', \mathscr{P}', \mathscr{Q}) \triangleq (h * h', \mathscr{P} + \mathscr{P}', \mathscr{Q})$$
$$\mathscr{R} \leq \mathscr{R}' \quad \text{iff} \quad \mathscr{R}_{\mathsf{hp}} \leq \mathscr{R}'_{\mathsf{hp}},\ \mathscr{R}_{\mathsf{loc}} \leq \mathscr{R}'_{\mathsf{loc}} \text{ and } \mathscr{R}'_{\mathsf{glo}} = \mathscr{R}_{\mathsf{glo}}$$

Division by 2 is defined by $\frac{1}{2}(h, \mathscr{P}, \mathscr{Q}) = (h, \frac{1}{2}\mathscr{P}, \mathscr{Q})$. We note that $*$ is commutative, associative and monotone with respect to $\leq$, that $\mathscr{R} \leq \mathscr{R}'$ iff $\mathscr{R}' = \mathscr{R} * \mathscr{R}''$ for some $\mathscr{R}''$, and that least upper bounds of bounded resource sets exist and are computed componentwise. For heap h and global permission table $\mathscr{Q}$, we define the subheap of h that consists of all its `final` fields:

$$h|\mathscr{Q} \triangleq /\!/\{\ (o,T,x) \in \lVert h \mid x = \bot \text{ or } \mathscr{Q}(x) < 1\ \} \qquad \mathsf{final}(h, \mathscr{P}, \mathscr{Q}) \triangleq (h|\mathscr{Q}, \mathbf{0}, \mathscr{Q})$$

If $\mathsf{final}(\mathscr{R}_i) = \mathsf{final}(\mathscr{R}_j)$ for all i, j then the greatest lower bound of $\{\mathscr{R}_i \mid i \in I\}$ exists.

4.2 Predicate Environments

For the semantics of predicates, we need a function that maps predicate symbols to relations. This function is called a *predicate environment*. We choose to represent relations as functions into the two-element set: Let $\mathbb{2}$ be the two-element set $\{0,1\}$ equipped with the usual order (i.e., $0 \leq 1$). Clearly, $\mathbb{2}$ is a complete lattice. The order

on sets $X \to L$ of total functions from X into complete lattice L is defined pointwise: $f \leq g$ whenever $f(x) \leq g(x)$ for all x in X. Clearly, $X \to L$ is a complete lattice where greatest lower bounds are computed pointwise.

The following shorthands are convenient:

$$\mathsf{SpecVals} \triangleq \bigcup_{n \geq 0} \mathsf{SpecVal}^n \quad \mathsf{Pred}(ct) \triangleq \{\ P@C \mid C \in \mathsf{dom}(ct) \text{ and } P \text{ is defined in } C\ \}$$

Predicate symbols are interpreted as relations over $\mathsf{SpecVals} \times \mathsf{Resources} \times \mathsf{ObjId} \times \mathsf{SpecVals}$. The first component represents the class parameters, the third component the receiving object and the fourth component the predicate parameters. For each predicate symbol $\kappa \in \mathsf{Pred}(ct)$, we define its *domain* $\mathsf{Dom}(\kappa)$: $(\overline{\pi}, \mathscr{R}, r, \overline{\pi'}) \in \mathsf{Dom}(P@C)$ iff all of the following statements hold:

(a) $\mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash r : C\langle\overline{\pi}\rangle$.
(b) $\mathsf{ptype}(P, C\langle\overline{\pi}\rangle) = \mathit{fin}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle$ and $\mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \overline{\pi'} : \overline{T}$ for some $\mathit{fin}, \mathit{pmod}, \overline{T}, \overline{\alpha}$.

This definition makes uses of a typing judgment $\Gamma \vdash \pi : T$ ("π has type T") where Γ is a function from $\mathsf{ObjId} \cup \mathsf{Var}$ to Ty. We omit the (obvious) typing rules. The partial function $\mathsf{ptype}(P, C\langle\overline{\pi}\rangle)$ looks up the type of predicate P in the least supertype of $C\langle\overline{\pi}\rangle$ that defines or extends P.

Definition 2 (Predicate Environments) A *predicate environment* is a function of type $\prod \kappa \in \mathsf{Pred}(ct).\mathsf{Dom}(\kappa) \to \mathbb{2}$ such that the following axioms hold:

(a) If $(\overline{\pi}, \mathscr{R}, r, \overline{\pi'}), (\overline{\pi}, \mathscr{R}', r, \overline{\pi'}) \in \mathsf{Dom}(\kappa)$ and $\mathscr{R} \leq \mathscr{R}'$,
then $\mathscr{E}(\kappa)(\overline{\pi}, \mathscr{R}, r, \overline{\pi'}) \leq \mathscr{E}(\kappa)(\overline{\pi}, \mathscr{R}', r, \overline{\pi'})$.
(b) If $(\overline{\pi}, \mathscr{R}, r, \overline{\pi'}) \in \mathsf{Dom}(\kappa)$ and $\mathsf{final}(\mathscr{R}_i) = \mathsf{final}(\mathscr{R}_j)$ for all i, j in I,
then $\mathscr{E}(P^{\mathsf{grp}}@C)(\overline{\pi}, \bigwedge_{i \in I} \mathscr{R}_i, r, \overline{\pi'}) = \bigwedge_{i \in I} \mathscr{E}(P^{\mathsf{grp}}@C)(\overline{\pi}, \mathscr{R}_i, r, \overline{\pi'})$.
(c) If $(\overline{\pi}, \mathscr{R}, r, \overline{\pi'}) \in \mathsf{Dom}(P^{\mathsf{grp}}@C)$,
then $\mathscr{E}(P^{\mathsf{grp}}@C)(\overline{\pi}, \mathscr{R}, r, \overline{\pi'}) \leq \mathscr{E}(P^{\mathsf{grp}}@C)(\overline{\pi}, \frac{1}{2}\mathscr{R}, r, \mathsf{split}(\overline{\pi'}))$.
(d) If $(\overline{\pi}, \mathscr{R}_1, r, \overline{\pi'}), (\overline{\pi}, \mathscr{R}_2, r, \overline{\pi'}) \in \mathsf{Dom}(P^{\mathsf{grp}}@C)$,
then $\mathscr{E}(P^{\mathsf{grp}}@C)(\overline{\pi}, \mathscr{R}_1, r, \mathsf{split}(\overline{\pi'})) \wedge \mathscr{E}(P^{\mathsf{grp}}@C)(\overline{\pi}, \mathscr{R}_2, r, \mathsf{split}(\overline{\pi'})) \leq \mathscr{E}(P^{\mathsf{grp}}@C)$
$(\overline{\pi}, \mathscr{R}_1 * \mathscr{R}_2, r, \overline{\pi'})$.
(e) If $(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi'}), (\overline{\pi}, (h', \mathscr{P}, \mathscr{Q}'), r, \overline{\pi'}) \in \mathsf{Dom}(\kappa)$, $o \in \mathsf{dom}(h)$, $\mathscr{P}(o, f) =$
0, $\mathscr{Q}(o, f) = 1$, $\mathscr{Q}' = \mathscr{Q}[(o, f) \mapsto 0]$ and $h' = h[o.f \mapsto v]$,
then $\mathscr{E}(\kappa)(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi'}) \leq \mathscr{E}(\kappa)(\overline{\pi}, (h', \mathscr{P}, \mathscr{Q}'), r, \overline{\pi'})$.
(f) If $(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi'}), (\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}'), r, \overline{\pi'}) \in \mathsf{Dom}(\kappa)$, $o \in \mathsf{dom}(h)$, $\mathscr{P}(o, \mathtt{join}) \leq$
$x \leq \mathscr{Q}(o, \mathtt{join})$, and $\mathscr{Q}' = \mathscr{Q}[(o, \mathtt{join}) \mapsto x]$,
then $\mathscr{E}(\kappa)(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi'}) \leq \mathscr{E}(\kappa)(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}'), r, \overline{\pi'})$.

Axiom (a) says that predicates are monotone in the resources: if a predicate is satisfied in resource $\mathscr{R}$ then it is also satisfied in all larger resources $\mathscr{R}'$. This axiom is natural for a language with garbage collection and is also imposed by Parkinson/Bierman's seminal work on abstract predicates [27]. The other axioms are new. Axioms (b), (c) and (d) are specific to data groups. Axiom (b) implies that data groups have a minimal satisfying resource. In separation logic terminology, one would say that data groups are *supported*. In order to ensure that data group implementations really satisfy this property, we will restrict the class of formulas that may implement data groups. Axiom (c) says that splitting the parameters of a data group splits the data group itself in half: if $\mathscr{R}$ satisfies $o.P@C\langle\overline{\pi}\rangle$, then $\frac{1}{2}\mathscr{R}$ satisfies $o.P@C\langle\mathsf{split}(\overline{\pi})\rangle$. Axiom (d) says that

merging two half data group parameters into a whole, merges the data group itself: if $\mathscr{R}_1$ and $\mathscr{R}_2$ both satisfy $o.P@C\texttt{<}\mathsf{split}(\overline{\pi})\texttt{>}$ then $\mathscr{R}_1 * \mathscr{R}_2$ satisfies $o.P@C\texttt{<}\overline{\pi}\texttt{>}$. Axioms (e) and (f) are technical conditions used to update the global permission table: (e) when fields get finalized and (f) when threads get joined.

It is easy to verify that the axioms for predicate environments are closed under taking pointwise infima. Thus, $\mathsf{Pred}(ct)$ *is a complete lattice* with respect to the pointwise order inherited from the underlying function space.

4.3 Kripke Resource Semantics

We define a relation "$\mathscr{Q};h;s \models \mathsf{pure}(e)$". Intuitively, $\mathscr{Q};h;s \models \mathsf{pure}(e)$ holds whenever $\mathscr{Q}(o,f) < 1$ for all heap cells $o.f$ that $[\![e]\!]^h_s$ depends on. Formally, the relation is defined by induction on the structure of e. We omit the routine definition of the partial function $[\![e]\!]^h_s$ that interprets expression e in heap h and stack s.

$$
\begin{array}{lcll}
\mathscr{Q};h;s & \models & \mathsf{pure}(\pi) & \\
\mathscr{Q};h;s & \models & \mathsf{pure}(\ell) & \\
\mathscr{Q};h;s & \models & \mathsf{pure}(op(\overline{e})) & \text{iff} \quad (\forall e \in \overline{e})(\mathscr{Q};h;s \models \mathsf{pure}(e)) \\
\mathscr{Q};h;s & \models & \mathsf{pure}(e.f) & \text{iff} \quad \mathscr{Q};h;s \models \mathsf{pure}(e), [\![e]\!]^h_s = o \text{ and } \mathscr{Q}(o,f) < 1
\end{array}
$$

The *semantic validity relation* $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models F)$ is the unique well-typed relation that satisfies the following statements (where Γ_{hp} is the restriction of Γ to ObjId, and $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$ iff $\Gamma'_{\mathsf{hp}} \supseteq \Gamma_{\mathsf{hp}}$ and $\Gamma'_{|\mathsf{Var}} = \Gamma_{|\mathsf{Var}}$):

$$
\begin{array}{lcll}
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{Q});s & \models & e & \text{iff} \quad \mathscr{Q};h;s \models \mathsf{pure}(e) \text{ and } [\![e]\!]^h_s = \texttt{true} \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{Q});s & \models & \texttt{PointsTo(}e\texttt{[}f\texttt{],}\pi\texttt{,}e'\texttt{)} & \text{iff} \quad \left\{ \begin{array}{l} \mathscr{Q};h;s \models \mathsf{pure}(e,e'), [\![e]\!]^h_s = o, \\ [\![\pi]\!] \leq \mathscr{P}(o,f) \text{ and } h(o)_2(f) = [\![e']\!]^h_s \end{array} \right. \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{Q});s & \models & \texttt{Perm(}e\texttt{[join],}\pi\texttt{)} & \text{iff} \quad \left\{ \begin{array}{l} \mathscr{Q};h;s \models \mathsf{pure}(e), [\![e]\!]^h_s = o \\ \text{and } [\![\pi]\!] \leq \mathscr{P}(o,\texttt{join}) \end{array} \right. \\
\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{Q});s & \models & \texttt{Pure(}e\texttt{)} & \text{iff} \quad \mathscr{Q};h;s \models \mathsf{pure}(e) \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & \texttt{null.}\kappa\texttt{<}\overline{\pi}\texttt{>} & \text{iff} \quad \text{true} \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & o.P@C\texttt{<}\overline{\pi}\texttt{>} & \text{iff} \quad \left\{ \begin{array}{l} \mathscr{R}_{\mathsf{hp}}(o)_1 <: C\texttt{<}\overline{\pi}'\texttt{>} \text{ and} \\ \mathscr{E}(P@C)(\overline{\pi}',\mathscr{R},o,\overline{\pi}) = 1 \end{array} \right. \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & o.P\texttt{<}\overline{\pi}\texttt{>} & \text{iff} \quad \left\{ \begin{array}{l} (\exists \overline{\pi}'')(\mathscr{R}_{\mathsf{hp}}(o)_1 = C\texttt{<}\overline{\pi}'\texttt{>} \text{ and} \\ \mathscr{E}(P@C)(\overline{\pi}',\mathscr{R},o,(\overline{\pi},\overline{\pi}'')) = 1) \end{array} \right. \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & F * G & \text{iff} \quad \left\{ \begin{array}{l} (\exists \mathscr{R}_1,\mathscr{R}_2)(\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2, \\ \Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models F \text{ and } \Gamma \vdash \mathscr{E};\mathscr{R}_2;s \models G) \end{array} \right. \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & F \mathrel{-\!\!*} G & \text{iff} \quad \left\{ \begin{array}{l} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathscr{R}')(\\ \mathscr{R} \# \mathscr{R}' \text{ and } \Gamma' \vdash \mathscr{E};\mathscr{R}';s \models F \\ \Rightarrow\ \Gamma' \vdash \mathscr{E};\mathscr{R} * \mathscr{R}';s \models G\ ) \end{array} \right. \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & F \,\&\, G & \text{iff} \quad \Gamma \vdash \mathscr{E};\mathscr{R};s \models F \text{ and } \Gamma \vdash \mathscr{E};\mathscr{R};s \models G \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & F \mid G & \text{iff} \quad \Gamma \vdash \mathscr{E};\mathscr{R};s \models F \text{ or } \Gamma \vdash \mathscr{E};\mathscr{R};s \models G \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & \texttt{(ex }T\ \alpha\texttt{)(}F\texttt{)} & \text{iff} \quad \left\{ \begin{array}{l} (\exists \pi)(\ \Gamma_{\mathsf{hp}} \vdash \pi : T \text{ and} \\ \Gamma \vdash \mathscr{E};\mathscr{R};s \models F[\pi/\alpha]\ ) \end{array} \right. \\
\Gamma \vdash \mathscr{E};\mathscr{R};s & \models & \texttt{(fa }T\ \alpha\texttt{)(}F\texttt{)} & \text{iff} \quad \left\{ \begin{array}{l} (\forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, \mathscr{R}' \geq \mathscr{R}, \pi)(\\ \Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond \text{ and } \Gamma'_{\mathsf{hp}} \vdash \pi : T \\ \Rightarrow\ \Gamma' \vdash \mathscr{E};\mathscr{R}';s \models F[\pi/\alpha]\ ) \end{array} \right.
\end{array}
$$

4.4 Predicate Definitions

We need to relate abstract predicate environments to the predicate implementations in the class table. This is a little tricky, because predicate definitions can be recursive. We therefore define a class table *ct*'s predicate environment as the least fixed point of a functional $\mathscr{F}_{ct}$.

$$\frac{\begin{array}{l}\mathsf{pbody}(o.P\texttt{<}\overline{\pi}'\texttt{>}, C\texttt{<}\overline{\pi}\texttt{>}) = F \texttt{ ext } D\texttt{<}\overline{\pi}''\texttt{>} \\ C \neq \texttt{Object} \text{ and } \mathsf{arity}(P,D) = n \;\Rightarrow\; F' = o.P@D\texttt{<}\overline{\pi}'_{to\,n}\texttt{>} \\ C = \texttt{Object} \text{ or } P \text{ is rooted in } C \;\Rightarrow\; F' = \texttt{true}\end{array}}{\mathscr{F}_{ct}(\mathscr{E})(P@C)(\overline{\pi}, \mathscr{R}, o, \overline{\pi}') = \begin{cases} 1 & \text{if } \mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \mathscr{E}; \mathscr{R}; \emptyset \models F * F' \\ 0 & \text{otherwise} \end{cases}}$$

Here, $\mathsf{pbody}(o.P\texttt{<}\overline{\pi}'\texttt{>}, C\texttt{<}\overline{\pi}\texttt{>})$ looks up $o.P\texttt{<}\overline{\pi}'\texttt{>}$'s definition in the type $C\texttt{<}\overline{\pi}\texttt{>}$ and returns its body F together with $C\texttt{<}\overline{\pi}\texttt{>}$'s direct superclass $D\texttt{<}\overline{\pi}''\texttt{>}$.

In order to guarantee that $\mathscr{F}_{ct}$ has a fixed point, we require that all cycles in the predicate dependency graph consist of positive dependencies only. ($P@C$ depends *negatively* on $P@D$, if $P@D$ occurs in $P@C$'s definition as the left descendant of an odd number of implications.) See Appendix L.10 for details.

Now we can state the partial correctness theorem:

Theorem 3 (Partial Correctness)
If $(ct, c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow_{ct}^{*} \langle h, ts \mid o \,\mathsf{is}\, (s \,\mathsf{in}\, \texttt{assert}(F); c)\rangle$, *then* $(\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{Q}); s \models F[\sigma])$ *for some* $\Gamma, \mathscr{E} = \mathscr{F}_{ct}(\mathscr{E}), \mathscr{P}, \mathscr{Q}$ *and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

5 Proof Theory

Ultimately, we are interested in algorithmic verification. For this reason, we do not want to base our Hoare logic on a Kripke semantics, but instead use a proof-theoretic logical consequence relation, as this seems more amenable to automation.

Our *logical consequence judgment* has the following forms:

$\Gamma; v; \overline{F} \vdash G$	from v's point of view, G is a logical consequence of the $*$-conjunction of $\overline{F}$
$\Gamma; v \vdash F$	from v's point of view, F is an axiom

In the former judgment, $\overline{F}$ is a *multiset* of formulas. The parameter v represents the *receiver*. The receiver parameter is needed to determine the scope of predicate definitions: a receiver v knows the definitions of predicates of the form $v.P$, but not the definitions of other predicates.

These main judgments depend on three auxiliary judgments, namely, *semantic validity of boolean expressions* $\Gamma \models e$ ("e is valid in all well-typed stack/heap pairs"), *syntactic purity judgments* $\overline{F} \vdash e : \checkmark$ ("if all expressions occurring in $\overline{F}$ are pure, then e is pure"), and $\overline{F} \vdash G : \checkmark$ ("if all expressions occurring in $\overline{F}$ are pure, then all expressions occurring in G are pure"). The latter two judgments are needed because in our system the set of pure expressions grows dynamically when fields are declared `final`.

Here are the technical definitions of the syntactic purity judgments: Let $\overline{F} \vdash e : \checkmark$ iff all field selection expressions $e'.f$ that occur in e also occur in $\overline{F}$; let $\overline{F} \vdash G : \checkmark$ iff $\overline{F} \vdash e : \checkmark$ for all expressions e that occur in G.[3]

[3] This definition is invariant under α-conversions, because bound variables in F must not occur in field selection expressions by syntactic restriction.

The logical consequence judgment is driven by natural deduction rules that are common to the logic of bunched implications [25] and linear logic [31]. In addition, there is a special introduction rule for the predicate $\texttt{Pure}(e)$ and a number of axioms that describe properties of our particular application domain. We admit weakening, because we do not want to reason about memory leaks, as Java is a garbage-collected language.

Logical Consequence, $\Gamma; v; \bar{F} \vdash G$**:**

$$\frac{\Gamma; v \vdash \bar{F}, G : \diamond}{\Gamma; v; \bar{F}, G \vdash G}\ (\text{Id}) \qquad \frac{\Gamma; v \vdash G \quad \Gamma; v \vdash \bar{F}, G : \diamond \quad \bar{F} \vdash G : \checkmark}{\Gamma; v; \bar{F} \vdash G}\ (\text{Ax}) \qquad \frac{\Gamma; v; \bar{F} \vdash G \quad \bar{F} \vdash e : \checkmark \quad \Gamma \vdash e : T}{\Gamma; v; \bar{F} \vdash G * \texttt{Pure}(e)}\ (\texttt{Pure Intro})$$

And the usual natural deduction rules for the other logical operators.

An important axiom is the *split/merge axiom*, which allows to split and merge fractional permissions. In order to formulate this axiom, we homomorphically extend the syntactic `split`-operator to arbitrary formulas:

$$\begin{gathered}
\mathsf{split}(e) \triangleq e \qquad \mathsf{split}(\texttt{Pure}(e)) \triangleq \texttt{Pure}(e) \qquad \mathsf{split}(\pi.\kappa\texttt{<}\bar{\pi}\texttt{>}) \triangleq \pi.\kappa\texttt{<}\mathsf{split}(\bar{\pi})\texttt{>} \\
\mathsf{split}(\texttt{PointsTo}(e\texttt{[}f\texttt{]}, \pi, e')) \triangleq \texttt{PointsTo}(e\texttt{[}f\texttt{]}, \mathsf{split}(\pi), e') \\
\mathsf{split}(\texttt{Perm}(e\texttt{[join]}, \pi)) \triangleq \texttt{Perm}(e\texttt{[join]}, \mathsf{split}(\pi)) \\
\mathsf{split}(F \mathit{\ lop\ } G) \triangleq \mathsf{split}(F) \mathit{\ lop\ } \mathsf{split}(G) \qquad \mathsf{split}(\texttt{(}qt\ T\ \alpha\texttt{)(}F\texttt{)}) \triangleq \texttt{(}qt\ T\ \alpha\texttt{)(}\mathsf{split}(F)\texttt{)}
\end{gathered}$$

The *split/merge axiom* is now formulated like this:

$$\Gamma; v \vdash F : \mathsf{supp} \ \Rightarrow\ \Gamma; v \vdash F \mathrel{\texttt{*-*}} \texttt{(}\mathsf{split}(F) \texttt{*} \mathsf{split}(F)\texttt{)}$$

The judgment $(\Gamma; v \vdash F : \mathsf{supp})$ ("F is supported") will be defined in Section 6.3. We note here that the `PointsTo`-predicate and boolean expressions are supported. Thus, as an instance of split/merge we get $e \mathrel{\texttt{*-*}} \texttt{(}e \texttt{*} e\texttt{)}$. This means that boolean expressions are copyable; they never get consumed.

For the following axioms, recall that "F `assures` G" abbreviates "$F \mathrel{\texttt{-*}} \texttt{(}F \texttt{*} G\texttt{)}$".

$$\begin{gathered}
\Gamma; v \vdash \texttt{true} \qquad \Gamma; v \vdash \texttt{false} \mathrel{\texttt{-*}} F \\
\Gamma; v \vdash \texttt{(PointsTo(}e\texttt{[}f\texttt{]}, \pi, e'\texttt{) \& PointsTo(}e\texttt{[}f\texttt{]}, \pi', e''\texttt{))} \ \texttt{assures}\ e' \texttt{==} e'' \\
(\Gamma \vdash e : T) \ \Rightarrow\ \Gamma; v \vdash \texttt{Pure(}e\texttt{)} \mathrel{\texttt{-*}} \texttt{(ex}\ T\ \alpha\texttt{)(}e \texttt{==} \alpha\texttt{)} \\
(\Gamma \vdash e, e' : T \ \wedge\ \Gamma, x : T \vdash F : \diamond) \Rightarrow \Gamma; v \vdash (F[e/x] \texttt{*} e \texttt{==} e') \mathrel{\texttt{-*}} F[e'/x]
\end{gathered}$$

The first of the following axioms lifts semantic validity of boolean expressions to our proof theory. The second axiom allows to apply class axioms. Here, $\mathsf{axiom}(t\texttt{<}\bar{\pi}'\texttt{>})$ is the $*$-conjunction of all class axioms in $t\texttt{<}\bar{\pi}'\texttt{>}$ and its supertypes.

$$\begin{gathered}
(\Gamma \models \texttt{!}e_1 \texttt{ | !}e_2 \texttt{ | } e') \ \Rightarrow\ \Gamma; v \vdash (e_1 \texttt{*} e_2) \mathrel{\texttt{-*}} e' \\
(\Gamma \vdash \pi : t\texttt{<}\bar{\pi}'\texttt{>} \ \wedge\ \mathsf{axiom}(t\texttt{<}\bar{\pi}'\texttt{>}) = F) \ \Rightarrow\ \Gamma; v \vdash F[\pi/\texttt{this}]
\end{gathered}$$

What is missing are the axioms for abstract predicates. The first of these axioms allows predicate receivers to replace their own abstract predicates by their definitions:

$$\begin{gathered}
(\Gamma \vdash v : C\texttt{<}\bar{\pi}''\texttt{>} \ \wedge\ \mathsf{pbody}(v.P\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F \ \mathsf{ext}\ D\texttt{<}\bar{\pi}'''\texttt{>}) \\
\Rightarrow\ \Gamma; v \vdash v.P\texttt{@}C\texttt{<}\bar{\pi}, \bar{\pi}'\texttt{>} \mathrel{\texttt{*-*}} \texttt{(}F \texttt{*} v.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}\texttt{)}
\end{gathered}$$

Note that the current receiver, as represented on the left of the $\vdash$, has to match the predicate receiver on the right. This rule is the only reason why our logical consequence judgment tracks the current receiver. Note also that $P@C$ may have a higher arity than $P@D$: following Parkinson/Bierman [27] we allow subclasses to extend predicate arities.

The following axioms capture some facts about abstract predicates that are always true, making crucial use of the `ispartof` derived from.

$$\Gamma; v \vdash \texttt{null}.\kappa\texttt{<}\bar{\pi}\texttt{>} \qquad \Gamma; v \vdash \pi.P@\texttt{Object} \qquad \Gamma; v \vdash \pi.P@C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{<}\bar{\pi}\texttt{>}$$
$$C \preceq D \ \Rightarrow\ \Gamma; v \vdash \pi.P@D\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P@C\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>}$$

The next axiom deals with predicates with missing parameters (resulting from arity extensions in subclasses). The missing parameters are existentially quantified.

$$\Gamma; v \vdash \pi.P\texttt{<}\bar{\pi}\texttt{>}\ \texttt{*-*}\ \texttt{(ex}\ \bar{T}\ \bar{\alpha}\texttt{)(}\pi.P\texttt{<}\bar{\pi},\bar{\alpha}\texttt{>)}$$

Finally, there are axioms to drop the class modifier C from $\pi.P@C$ if we know that C is π's dynamic class.

$$\Gamma; v \vdash \texttt{(}\ \pi.P@C\texttt{<}\bar{\pi}\texttt{>} * C\ \texttt{isclassof}\ \pi\ \texttt{)}\ \texttt{-*}\ \pi.P\texttt{<}\bar{\pi}\texttt{>}$$
$$(C \text{ is } \texttt{final} \text{ or } P \text{ is } \texttt{final} \text{ in } C) \ \Rightarrow\ \Gamma; v \vdash \pi.P@C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{-*}\ \pi.P\texttt{<}\bar{\pi}\texttt{>}$$

We note that our treatment of subclassing for abstract predicates differs from Parkinson/Bierman [27] in that it formalizes the "stack of class frames" from Fähndrich/DeLine's typestate system [13] (also present in the Boogie methodology [3]). The advantage of the stack of class frames is that it enables full modularity, whereas Parkinson/Bierman have to reverify inherited methods. We find that our separation logic explanation of the stack of class frame, using the `ispartof` predicate, is quite concise.

To state soundness for our proof theory, we define semantic entailment. First, we define a semantic counterpart to the syntactic purity judgment:[4]

$$\mathscr{Q}; h; s \models \bar{F} : \checkmark \quad \text{iff} \quad \mathscr{Q}; h; s \models \mathsf{pure}(e) \text{ for all field selection subexpressions } e \text{ of } \bar{F}$$

Now, we define *semantic entailment*. (As usual, we let $F_1 * \cdots * F_0 = \texttt{true}$.)

$$\begin{array}{rll}
\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F : \checkmark & \text{iff} & (\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F \text{ and } \mathscr{R}_{\mathsf{glo}}; \mathscr{R}_{\mathsf{hp}}; s \models F : \checkmark) \\
\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F_1, \ldots, F_n : \checkmark & \text{iff} & \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F_1 * \cdots * F_n : \checkmark \\
\Gamma \vdash \mathscr{E}; \bar{F} \models G : \checkmark & \text{iff} & (\forall \Gamma, \mathscr{R}, s)(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} : \checkmark \ \Rightarrow\ \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G : \checkmark)
\end{array}$$

Theorem 4 (Soundness of Logical Consequence) *If* $(\Gamma; o; \bar{F} \vdash G)$ *and* $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, *then* $(\Gamma \vdash \mathscr{E}; \bar{F} \models G : \checkmark)$.

6 The Verification System

6.1 Method Types and Predicate Types

Method types are of the following form:

$$MT \in \mathsf{MethTy} \ ::= \ \texttt{<}\bar{T}\ \bar{\alpha}\texttt{>req}\ F\texttt{; ens}\ G\texttt{;}\ U\ m\texttt{(}\bar{V}\ \bar{\imath}\texttt{)} \quad \text{(scope of } \bar{\alpha}, \bar{\imath} \text{ is } \bar{T}, F, G, U, \bar{V}\text{)}$$

[4] This definition is invariant under α-conversions, because bound variables in $\bar{F}$ must not occur in field selection expressions by syntactic restriction.

The function $\mathtt{mtype}(m, C\texttt{<}\bar{\pi}\texttt{>})$ looks up m's type in the smallest superclass of $C\texttt{<}\bar{\pi}\texttt{>}$ that declares m. It also replaces a post-condition G found in the program text, by $(\mathtt{ex}\, U\, \mathtt{result})(G)$ and makes the self-parameter explicit as the first method parameter. Thus, method types MT always have at least one parameter. The methods `fork` and `join` are special cases: $\mathtt{mtype}(\mathtt{fork}, C\texttt{<}\bar{\pi}\texttt{>})$ takes `run`'s precondition as the precondition and `true` as the postcondition; $\mathtt{mtype}(\mathtt{join}, C\texttt{<}\bar{\pi}\texttt{>})$ takes `run`'s postcondition as the postcondition and `true` as the precondition.

Our notion of method subtyping generalizes behavioral subtyping. It is defined in terms of proof-theoretic logical consequence:

Method Subtyping, $\Gamma \vdash MT <: MT'$ and $\Gamma \vdash \mathit{fin}\, MT <: \mathit{fin}'\, MT'$:

$$\frac{m \neq \mathtt{run} \quad \bar{T}' <: \bar{T} \quad U <: U' \quad V_0 <: V_0' \quad \bar{V}' <: \bar{V} \quad G = (\mathtt{ex}\, U\, \alpha'')(G_0) \quad G' = (\mathtt{ex}\, U'\, \alpha'')(G_0') \quad \Gamma, \iota_0 : V_0; \iota_0; \mathtt{true} \vdash (\mathtt{fa}\, \bar{T}'\, \bar{\alpha})(\mathtt{fa}\, \bar{V}'\, \bar{\iota})(F' \mathtt{-\!*} (\mathtt{ex}\, \bar{W}\, \bar{\alpha}')(F * (\mathtt{fa}\, U\, \alpha'')(G_0 \mathtt{-\!*} G_0')))}{\Gamma \vdash \texttt{<}\bar{T}\, \bar{\alpha}, \bar{W}\, \bar{\alpha}'\texttt{>}\mathtt{req}\, F; \mathtt{ens}\, G; U\, m(V_0\, \iota_0, \bar{V}\, \bar{\iota}) <: \texttt{<}\bar{T}'\, \bar{\alpha}\texttt{>}\mathtt{req}\, F'; \mathtt{ens}\, G'; U'\, m(V_0'\, \iota_0, \bar{V}'\, \bar{\iota})} \text{ (Mth Sub)}$$

$$\frac{G = (\mathtt{ex}\, \mathtt{void}\, \alpha)(G_0) \quad G' = (\mathtt{ex}\, \mathtt{void}\, \alpha)(G_0') \quad V_0 <: V_0' \quad \Gamma, \iota_0 : V_0; \iota_0; \mathtt{true} \vdash (F' \mathtt{-\!*} F) * (\mathtt{fa}\, \mathtt{void}\, \alpha)(G_0 \mathtt{-\!*} G_0')}{\Gamma \vdash \mathtt{req}\, F; \mathtt{ens}\, G; \mathtt{void}\, \mathtt{run}(V_0\, \iota_0) <: \mathtt{req}\, F'; \mathtt{ens}\, G'; \mathtt{void}\, \mathtt{run}(V_0'\, \iota_0)} \text{ (Run Sub)}$$

For qualified method types: $\Gamma \vdash \mathit{fin}\, MT <: \mathit{fin}'\, MT'$ iff $\mathit{fin}' = \varepsilon \wedge \Gamma \vdash MT <: MT'$

Example 1 (Behavioral Subtyping) *If* $\bar{T}', U, V_0, \bar{V}' <: \bar{T}, U', V_0', \bar{V}$ *and* $G = (\mathtt{ex}\, U\, \alpha')(G_0)$ *and* $G' = (\mathtt{ex}\, U'\, \alpha')(G_0')$ *and* $\iota_0 : V_0; \iota_0; \mathtt{true} \vdash (\mathtt{fa}\, \bar{T}'\, \bar{\alpha})(\mathtt{fa}\, \bar{V}'\, \bar{\iota})((F' \mathtt{-\!*} F) * (\mathtt{fa}\, U\, \alpha')(G_0 \mathtt{-\!*} G_0'))$, *then* $\texttt{<}\bar{T}\, \bar{\alpha}\texttt{>}\mathtt{req}\, F; \mathtt{ens}\, G; U\, m(V_0\, \iota_0, \bar{V}\, \bar{\iota}) <: \texttt{<}\bar{T}'\, \bar{\alpha}\texttt{>}\mathtt{req}\, F'; \mathtt{ens}\, G'; U'\, m(V_0'\, \iota_0, \bar{V}'\, \bar{\iota})$.

Example 2 (Auxiliary Variable Elimination)

If $C <: D$ *and* $(\alpha : T \vdash F, G : \diamond)$, *then*
$\texttt{<}T\, \alpha\texttt{>}\mathtt{req}\, F; \mathtt{ens}\, G; \mathtt{void}\, m(C\, \iota) <: \mathtt{req}\, (\mathtt{ex}\, T\, \alpha)(F); \mathtt{ens}\, (\mathtt{ex}\, T\, \alpha)(G); \mathtt{void}\, m(D\, \iota)$.

Note that the left method type is not a behavioral subtype of the right one.

For predicates, we allow to extend their arity in subclasses. On the other hand, we disallow arity extensions for data groups, because the existential quantification over missing arguments would render the merge axiom for data groups unsound.

6.2 Hoare Triples

Our judgment for commands combines typing and Hoare triples. We present the judgment in an algorithmic style without structural rules. Hoare triples have the forms $(\Gamma; v \vdash \{F\}c : T\{G\})$ and $(\Gamma; v \vdash \{F\}hc\{G\})$, where in the former the postcondition G is always of the form $G = (\mathtt{ex}\, T\, \alpha)(G')$. In these judgments, v is the receiver parameter which is needed because the logical consequence judgment depends on it (in order to determine the scope of predicate definitions). In an implementation, the receiver parameter v could be omitted because in source code we always have $v = \mathtt{this}$. We make the receiver parameter explicit, because we want our judgments to be closed under value substitutions. We will write $(\Gamma; v \vdash F : \diamond)$ to abbreviate $\Gamma \vdash F : \diamond \wedge \Gamma \vdash v : \mathtt{Object}$.

Hoare Triples, $\Gamma;\nu \vdash \{F\}c : T\{G\}$ **and** $\Gamma;\nu \vdash \{F\}hc\{G\}$**:**

(Seq)
$$\frac{\Gamma;\nu;F \vdash F' \quad \Gamma;\nu \vdash \{F'\}hc\{E\} \quad \Gamma;\nu \vdash \{E\}c : T\{G\}}{\Gamma;\nu \vdash \{F\}hc;c : T\{G\}}$$

(Val)
$$\frac{\Gamma,\alpha : T \vdash G : \diamond \quad \Gamma \vdash w : T' <: T \quad \Gamma;\nu;F \vdash G[w/\alpha]}{\Gamma;\nu \vdash \{F\}w : T\{(\texttt{ex}\ T'\ \alpha)\ (G)\}}$$

(Dcl) $\ell \notin F,G$
$$\frac{\Gamma,\ell : T;\nu \vdash \{F * \ell == \mathsf{df}(T)\}c : U\{G\}}{\Gamma;\nu \vdash \{F\}T\ \ell;\ c : U\{G\}}$$

(Fin Dcl) $\iota \notin F,G,\nu$
$$\frac{\Gamma \vdash \ell : T \quad \Gamma,\iota : T;\nu \vdash \{F * \iota == \ell\}c : U\{G\}}{\Gamma;\nu \vdash \{F\}\texttt{final}\ T\ \iota = \ell;\ c : U\{G\}}$$

(Unpack) $\alpha \notin F,G$
$$\frac{\Gamma,\alpha : T;\nu \vdash \{F * E\}c : U\{G\}}{\Gamma;\nu \vdash \{F * (\texttt{ex}\ T\ \alpha)\ (E)\}\texttt{unpack}\ (\texttt{ex}\ T\ \alpha)\ (E);\ c : U\{G\}}$$

(Var Set)
$$\frac{\Gamma \vdash w : \Gamma(\ell) \quad \Gamma;\nu \vdash F : \diamond \quad \ell \notin F}{\Gamma;\nu \vdash \{F\}\ell = w\{F * \ell == w\}}$$

(Op)
$$\frac{\Gamma \vdash op(\overline{w}) : \Gamma(\ell) \quad \Gamma;\nu \vdash F : \diamond \quad \ell \notin F}{\Gamma;\nu \vdash \{F\}\ell = op(\overline{w})\{F * \ell == op(\overline{w})\}}$$

(Cast) $T <: \Gamma(\ell) <: \texttt{Object}$
$$\frac{\Gamma \vdash w : \texttt{Object} \quad \Gamma;\nu \vdash F,T : \diamond \quad \ell \notin F}{\Gamma;\nu \vdash \{F\}\ell = (T)w\{F * \ell == w\}}$$

(If)
$$\frac{\Gamma;\nu \vdash \{F * \texttt{!}w\}c' : \texttt{void}\{G\} \quad \Gamma \vdash w : \texttt{bool} \quad \Gamma;\nu \vdash \{F * w\}c : \texttt{void}\{G\}}{\Gamma;\nu \vdash \{F\}\texttt{if}\ (w)\{c\}\texttt{else}\{c'\}\{G\}}$$

(Assert)
$$\frac{\Gamma;\nu;F \vdash G}{\Gamma;\nu \vdash \{F\}\texttt{assert}(G)\{F\}}$$

(New) $C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} \in ct$
$$\frac{\Gamma \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}] \quad C\texttt{<}\overline{\pi}\texttt{>} <: \Gamma(\ell) \quad \Gamma;\nu \vdash F : \diamond \quad \ell \notin F}{\Gamma;\nu \vdash \{F\}\ell = \texttt{new}\ C\texttt{<}\overline{\pi}\texttt{>}\{F * \ell.\texttt{init} * C\ \texttt{isclassof}\ \ell\}}$$

(Get) $\Gamma;\nu;F \vdash F' \quad \Gamma \vdash w.f : \Gamma(\ell) \quad \ell \notin F$
$$\frac{(F',F'') = (\texttt{PointsTo}(w\texttt{[}f\texttt{]},\pi,u),\ \ell == u) \text{ or } (F',F'') = (\texttt{Pure}(w.f),\ \ell == w.f)}{\Gamma;\nu \vdash \{F\}\ell = w.f\{F * F''\}}$$

(Fld Set) $\Gamma \vdash u : C\texttt{<}\overline{\pi}\texttt{>} \quad T\ f \in \mathsf{fld}(C\texttt{<}\overline{\pi}\texttt{>}) \quad \Gamma \vdash w : T$
$$\frac{\Gamma;\nu \vdash F : \diamond \quad (\mathit{fin},F') = (\varepsilon,\ \texttt{PointsTo}(u\texttt{[}f\texttt{]},1,w)) \text{ or } (\mathit{fin},F') = (\texttt{final},\ u.f == w)}{\Gamma;\nu \vdash \{F * \texttt{PointsTo}(u\texttt{[}f\texttt{]},1,T)\}\mathit{fin}\ u.f = w\{F * F'\}}$$

(Call) $m = \texttt{join} \Rightarrow F' = \mathit{fr} \cdot \texttt{Perm}(u\texttt{[join]},1) \qquad m \neq \texttt{join} \Rightarrow \mathit{fr} = \mathsf{all}$
$$\frac{\begin{array}{c}\mathsf{mtype}(m,t\texttt{<}\overline{\pi}\texttt{>}) = \mathit{fin}\ \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{req}\ G;\texttt{ens}\ (\texttt{ex}\ U\ \alpha')\ (G');\ U\ m(t\texttt{<}\overline{\pi}\texttt{>}\iota_0,\overline{W}\ \overline{\iota}) \\ \Gamma;\nu \vdash F : \diamond \quad \ell \notin F \quad \sigma = (u/\iota_0,\overline{\pi}'/\overline{\alpha},\overline{w}/\overline{\iota}) \quad \Gamma \vdash u,\overline{\pi}',\overline{w} : t\texttt{<}\overline{\pi}\texttt{>},\overline{T}[\sigma],\overline{W}[\sigma] \quad U[\sigma] <: \Gamma(\ell)\end{array}}{\Gamma;\nu \vdash \{F * F' * u\ \texttt{!=null} * G[\sigma]\}\ell = u.m\texttt{<}\overline{\pi}'\texttt{>}(\overline{w})\{F * (\texttt{ex}\ U[\sigma]\ \alpha')\ (\alpha' == \ell * \mathit{fr} \cdot G'[\sigma])\}}$$

In the verification rule (Dcl), $\mathsf{df}(T)$ is the default value of type T. The ℓ.`init` predicate in (New), $*$-conjoins the predicates $\texttt{PointsTo}(\ell\texttt{[}f\texttt{]},1,\mathsf{df}(T))$ for all fields $T\ f$ in ℓ's class, plus $\texttt{Perm}(\ell\texttt{[join]},1)$ in case ℓ has type `Thread`. In the verification rule for `join`-calls (an instance of (Call)), fr ranges over *linear combinations*. These represent numbers of the forms 1 or $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$:

$$bit \in \{0,1\} \qquad bits ::= 1 \mid bit,bits \qquad \mathit{fr} \in \mathsf{BinFrac} ::= \mathsf{all} \mid \mathsf{fr}() \mid \mathsf{fr}(bits)$$

The scalar multiplication $\mathit{fr} \cdot F$ is defined as follows: $\mathsf{all} \cdot F = F$, $\mathsf{fr}() \cdot F = \texttt{true}$, $\mathsf{fr}(1) \cdot F = \mathsf{split}(F)$, $\mathsf{fr}(0,bits) \cdot F = \mathsf{fr}(bits) \cdot \mathsf{split}(F)$, and $\mathsf{fr}(1,bits) \cdot F = \mathsf{split}(F) * \mathsf{fr}(bits) \cdot \mathsf{split}(F)$. For instance, $\mathsf{fr}(1,0,1) \cdot F \ *\!\!-\!\!*\ (\mathsf{split}(F) * \mathsf{split}^3(F))$.

[Chris says: The condition $F' = \texttt{true}$ is not needed. F' can be anything, because the rule just drops F' and that is not harmful. I'd prefer to make as few modifications to our submitted paper as possible. There are already far too many that we needed to make to ensure soundness, uhm, ... BTW, the condition $\mathit{fr} = \mathsf{all}$ was needed because scalar multiplication only works properly for supported formulas, and we only require supportedness for the postcondition of run/join, but not for other methods.]

6.3 Supported Formulas, Data Group Formulas, Join Postconditions

Recall the split/merge axiom: $\Gamma; v \vdash F : \mathsf{supp} \;\Rightarrow\; \Gamma; v \vdash F \;\mathtt{*-*}\; (\mathsf{split}(F) * \mathsf{split}(F))$

We do not have space for the inductive definition of $(\Gamma; v \vdash F : \mathsf{supp})$: Put shortly, it says that F does not contain `-*`, `|`, `fa`, abstract predicates other than data groups, and that all *existentials in F have unique witnesses*:

$$\frac{\Gamma \vdash T : \diamond \quad \Gamma, \alpha : T; v \vdash F : \mathsf{supp} \quad \Gamma, \alpha : T, \alpha' : T; v; \mathtt{true} \vdash' F \;\&\; F[\alpha'/\alpha] \;\mathtt{-*}\; \alpha == \alpha'}{\Gamma; v \vdash (\mathtt{ex}\; T\; \alpha)(F) : \mathsf{supp}}$$

The $\vdash'$ in the premise of this rule is a restricted logical consequence judgement that disallows the application of the merge direction of split/merge (needed to prevent circularities in our soundness proofs).

We require that formulas that define data groups must be supported. In addition, data group formulas must be fully permission parameterized, i.e, they must not contain permission constants or permission variables that are bound by class parameters.

Postconditions for `join` are also required to be supported, but for those the unique witness proof must not use class axioms and opening/closing of abstract predicates. These conditions ensure that supportedness is closed under formula splitting. Supportedness of `join`'s postcondition is only needed to enable that multiple joiners of the same thread can share the joined thread's resources for concurrent reading. Supportedness of `join`'s postcondition is not needed in a language with parallel composition (because multiple joiners are not possible) or in a logic without fractional permissions (because concurrent reading is not possible).

We omit the judgment "$ct : \diamond$" for good class tables. For programs, we define $(ct, c) : \diamond$ iff $(ct : \diamond$ and $\mathsf{main} : \mathtt{Thread}; \mathsf{main} \vdash \{\mathtt{true}\}c : \mathtt{void}\{\mathtt{true}\})$.

7 Preservation

Good States, $st : \diamond$**, Good Thread Pools,** $\mathscr{R} \vdash ts : \diamond$**:**

$$\frac{(h, \mathscr{P}, \mathscr{Q}) \vdash ts : \diamond}{\langle h, ts\rangle : \diamond}\ (\text{State}) \qquad \frac{}{\mathscr{R} \vdash \emptyset : \diamond}\ (\text{Empty Pool}) \qquad \frac{\mathscr{R} \vdash t : \diamond \quad \mathscr{R}' \vdash ts : \diamond \quad \mathsf{dom}(\mathscr{R}_{\mathsf{hp}}) = \mathsf{dom}(\mathscr{R}'_{\mathsf{hp}})}{\mathscr{R} * \mathscr{R}' \vdash t \mid ts : \diamond}\ (\text{Cons Pool})$$

Let $\mathsf{post}(C\texttt{<}\bar{\pi}\texttt{>}, \mathtt{run})$ be `run`'s postcondition in $C\texttt{<}\bar{\pi}\texttt{>}$. Let $(\Gamma \vdash \sigma : \Gamma')$ whenever $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$, $\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma')$ and $(\Gamma[\sigma] \vdash \sigma(\alpha) : \Gamma'(\alpha)[\sigma])$ for all α in $\mathsf{dom}(\sigma)$. Let $\mathsf{cfv}(c) = \{\alpha \in \mathsf{fv}(c) \mid \alpha$ occurs in an obj. creation command $\ell = \mathtt{new}\, C\texttt{<}\bar{\pi}\texttt{>}\}$.

Good Threads, $\mathscr{R} \vdash t : \diamond$

$$\frac{\begin{array}{c}\mathscr{R}_{\mathsf{glo}}(o, \mathtt{join}) \leq [\![fr]\!] \quad fr = \mathsf{all} \text{ or } (\exists \mathscr{R}' \geq \mathscr{R})(\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models G[o/\mathtt{this}]) \\ \mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E} \quad \Gamma \vdash \sigma : \Gamma' \quad \Gamma, \Gamma' \vdash s : \diamond \quad \mathsf{cfv}(c) \cap \mathsf{dom}(\Gamma') = \emptyset \quad \mathsf{post}(h(o)_1, \mathtt{run}) = G \\ \Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma] : \checkmark \quad \Gamma, \Gamma'; r \vdash \{F\}c : \mathtt{void}\{fr \cdot G[o/\mathtt{this}]\}\end{array}}{\mathscr{R} \vdash o \;\mathsf{is}\; (s \;\mathsf{in}\; c) : \diamond}\ (\text{Thread})$$

Intuitively, the type environment Γ assigns types to object ids and read-write variables, and Γ' assigns types to logic variables that got introduced by existential unpacking.

Theorem 5 (Preservation) *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$, *then* $(st' : \diamond)$.

8 Comparison to Related Work and Conclusion

Our general contribution compared to type-based race condition checkers [1, 7] and logical verification systems [18] for concurrent Java-like languages is support for fork/join synchronization. Compared to permission-accounting separation logic [6, 11], our system supports object orientation, including subclassing, dynamic dispatch and data groups. In contrast to our work, [1, 7, 18, 6, 11] all support lock-synchronization, which we deliberately omit to focus on fork/join. Clearly, a general system will have to support both lock and fork/join synchronization, as well as combinations thereof.

The only other work that integrates separation logic into a Java-like language (albeit sequential) that we are aware of is by Parkinson and Bierman [26, 27]. We build on their work, using abstract predicates, but extend it to a concurrent language and combine abstract predicates with fractional permissions. To our knowledge the combination of abstract predicates and fractional permissions is novel.

Boyland and Retert [9] present a type-and-effect system that is closely related to separation logic. They use their system to explain the relation between write-effects and uniqueness.

Recent work by Bierhoff and Aldrich [5] combines typestates and permissions (including fractional ones) to specify object usage protocols. They do not treat concurrency. Like us, they use iterators as an example. However, they do not allow linear implication in pre- and postconditions. As a result, their usage protocol regulates access to the collection itself, but not access to the elements of the collection (i.e., they do not account for the dashed arrows). Consequently, in concurrent programs, their usage protocol would not prevent data races. Krishnaswami [19] presents a higher order separation logic specification of iterators that is related to ours. His iterator does not have a remove-method and his language is sequential.

Gotsman and others recently adapted concurrent separation logic to more realistic concurrency primitives, including fork/join [15]. They do not support concurrent read access. In particular, they do not support read-sharing of join's postconditions like we do. A bit surprisingly to us, they require that fork's precondition is a precise predicate.

Conclusion. We have presented a variant of concurrent separation logic with fractional permissions for a Java-like language with fork/join and proved its soundness. Interesting future work includes algorithmic checking and extension to handle lock synchronization.

Acknowledgments. We thank Marieke Huisman and Erik Poll for their continuous and very helpful feedback.

A Examples

A.1 A Simple Fork/Join Example

Our first example is a recursive computation of the *n*-th Fibonacci number that runs recursive calls in new threads. The example is taken from Lea's collection of patterns [20, §4.4.1.4]. Although the example is unrealistic because there are faster non-recursive algorithms to compute Fibonacci numbers, it nicely illustrates how our system works with `fork()` and `join()`.

[Chris says: I always start a new paragraph before and after small font, because Latex often messes up the spacing between lines otherwise. (Compare the Latex output. You'll notice that it looks slightly screwed if you don't start a new paragraph.)]

```
class Fib ext Thread {
```

```
  int number;

  req init;
  ens PointsTo(this[number],1,n) * Perm(this[join],1);
  void init(int n) { number = n; }

  req Perm(number,1);
  ens Perm(number,1);
  public void run() {
    if(!(number < 2)){
        Fib f1 = new Fib(); f1.init(number-1);
        Fib f2 = new Fib(); f2.init(number-2);
        f1.fork(); f2.fork();
        f1.join(); f2.join();
        number = f1.number + f2.number;
    }
  }
}
```

We do not verify the functional behavior of this algorithm because we would need extra machinery (such as axiomatizing the Fibonacci function) but we prove race freedom. Here is the proof outline for `run()`:

```
  { Perm(this[number],1) }
  (Because we have Perm(this[number],1), we can read this.number)
if(!(this.number < 2)){
    { Perm(this[number],1) }
  Fib f1 = new Fib();
    { Perm(this[number],1) * f1.init }
  f1.init(number-1);
    { Perm(this[number],1) * PointsTo(f1[number],1,number-1) *
      Perm(f1[join],1) }
  Fib f2 = new Fib();
    { Perm(this[number],1) * PointsTo(f1[number],1,number-1) *
      Perm(f1[join],1) * f2.init }
  f2.init(number-2);
    { Perm(this[number],1) * PointsTo(f1[number],1,number-1) *
      Perm(f1[join],1) * PointsTo(f2[number],1,number-2) *
      Perm(f2[join],1) }
    (We use the precondition of run() as the precondition for fork())
  f1.fork();
    { Perm(this[number],1) * Perm(f1[join],1) *
      PointsTo(f2[number],1,number-2) * Perm(f2[join],1) }
    (We use the precondition of run() as the precondition for fork())
  f2.fork();
    { Perm(this[number],1) * Perm(f1[join],1) * Perm(f2[join],1) }
    (We use the postcondition of run() as the postcondition for join().
     Because we have Perm(f1[join],1), we have full access to
     the postcondition of f1.run())
  f1.join();
    { Perm(this[number],1) * Perm(f1[number],1) * Perm(f2[join],1) }
```

```
    (We use the postcondition of run() as the postcondition for join().
     Because we have Perm(f2[join],1), we have full access to
     the postcondition of f2.run())
  f2.join();
    { Perm(this[number],1) * Perm(f1[number],1) * Perm(f2[number],1) }
    (Because we have Perm(this[number],1), we can write to this.number.
     In addition, because we have Perm(f1[number],1), we can read f1.number
     (and similarly for f2.number).)
  this.number = f1[number] + f2[number];
    { ditto }
    (Dropping some clauses)
    { Perm(this[number],1) }
}
```

A.2 An Example with Recursive and Overlapping Datagroups

Our next example is a linked list implementation of a class roster that collects student identifiers and associates them with grades. We design the roster interface so that multiple threads can concurrently read a roster. Moreover, when a thread updates the grades we allow another threads to concurrently read the student identifiers.

Objects of type `Roster` have two datagroups with two permission parameters each:

`ids_and_links<p,q>`	datagroup of student ids and links between the student entries `p` is the permission for the student ids `q` is the permission for the links
`grades_and_links<p,q>`	datagroup of grades and links between the student entries `p` is the permission for the grades `q` is the permission for the links

Note that the two datagroups overlap on the links. Here is the separation logic contract for `Rosters`:

```
interface Roster {
   group ids_and_links<perm p, perm q>;
   group grades_and_links<perm p, perm q>;
   axiom state<p> *-* (ids_and_links<p,p/2> * grades_and_links<p,p/2>)
   req grades_and_links<1,p> * ids_and_links<q,r>;
   ens grades_and_links<1,p> * ids_and_links<q,r>;
   void updateGrade(int id, int grade);
   req ids_and_links<p,q>; ens ids_and_links<p,q>;
   bool contains(int id);
}
```

Here are informal interpretations of the method contracts:

`updateGrade(id,grade)`: Requires write access to the grades and read access to the student ids and the links. (We omit the quantifiers over the logic variable `p`, `q`, `r` because they can be inferred.)

`contains(id)`: Requires read access to the student ids and the links. (We omit the quantifiers over the logic variables `p`, `q` because they can be inferred.)

Our implementation also contains an `init()`-method:

`init(id,grade,next):` Plays the role of a constructor. Requires write access to the fields of `this` (by precondition `init`) and write access to the state of `next` (by precondition `next.state<1>`). Ensures write access to the roster (by postcondition `state@RosterImpl<1>`).

Note that `init()`'s postcondition refers to the implementation class `RosterImpl`. This is the reason why we cannot specify it in the interface (as interfaces do not know about their implementations).

```
final class RosterImpl impl Roster {
  int id; int grade; Students next;
  group ids_and_links<perm p, perm q> =
    Perm(id,p) *
    (ex RosterImpl x)( PointsTo(next,q,x) * x.ids_and_links<p,q> );
  group grades_and_links<perm p,perm q> =
    Perm(grade,p) *
    (ex RosterImpl x)( PointsTo(next,q,x) * x.grades_and_links<p,q> );
  extends group state<perm p> by
    ids_and_links@RosterImpl<p,p/2> * grades_and_links@RosterImpl<p,p/2>;
  req init * next.state<1>; ens state<1>;
  void init(int id, int grade, Students next) {
    this.id = id; this.grade = grade; this.next = next;
  }
  req grades_and_links<1,p> * ids_and_links<q,r>;
  ens grades_and_links<1,p> * ids_and_links<q,r>;
  void updateGrade(int id, int grade) {
    if (this.id == id) { this.grade = grade; }
    else if (next != null) { next.updateGrade(id,grade); }
  }
  req ids_and_links<p,q>; ens ids_and_links<p,q>;
  bool contains(int id) {
    bool b = this.id==id; if(!b && next!=null){ b=next.contains(id); }; b
  }
}
```

A.3 A Usage Protocol for Iterators

We present a doubly-linked list implementation of the `Iterator` interface from Section 2.5. In this example, we use regular Java and wrap contracts into special comments in the style of JML [12]. We also use the field and predicate modifier `spec_public` as additional syntax sugar. (This modifier is also part of JML [21].)

- Declaring a (possibly private) field f `spec_public` introduces a singleton datagroup f`<`p,x`>`, where p is the access permission for this field and x is the value contained in f:

$$\texttt{spec\_public}\ T\ f \ \triangleq\ \begin{array}{l} T\ f; \\ \texttt{group}\ f\texttt{<perm}\ p, T\ x\texttt{>=PointsTo(this[}f\texttt{]},p,x\texttt{)}; \\ \texttt{axiom}\ (f\texttt{<}p,x\texttt{>}\ \&\ f\texttt{<}q,y\texttt{>})\ \texttt{assures}\ x == y \end{array}$$

- Declaring a predicate `spec_public` exports its definition as an axiom. For predicate definitions in class C:

$$\texttt{spec\_public}\ \textit{pmod}\ P\texttt{<}\overline{T}\ \overline{x}\texttt{>} = F \quad \triangleq \quad \begin{array}{l}\textit{pmod}\ P\texttt{<}\overline{T}\ \overline{x}\texttt{>} = F\texttt{;}\\ \texttt{axiom}\ P\texttt{@}C\texttt{<}\overline{x}\texttt{>}\ \texttt{*-*}\ F\end{array}$$

- Declaring a predicate extension `spec_public` exports the extension as an axiom. For predicate extensions in class C extending D:

$$\texttt{spec\_public extends}\ \textit{pmod}\ P\texttt{<}\overline{T}\ \overline{x}\texttt{>} = F \quad \triangleq \quad \begin{array}{l}\texttt{extends}\ \textit{pmod}\ P\texttt{<}\overline{T}\ \overline{x}\texttt{>} = F\texttt{;}\\ \texttt{axiom}\ P\texttt{@}C\texttt{<}\overline{x}\texttt{>}\ \texttt{*-*}\ (F\ \texttt{*}\ P\texttt{@}D\texttt{<}\overline{y}\texttt{>})\end{array}$$

  where $\overline{y}$ is the prefix of $\overline{x}$ that matches P@D's arity

A.3.1 The Collection Interface

```
interface Collection {

  //@ req init; ens state<1>;
  void init();

  //@ req state<1> * x.state<1>; ens state<1>;
  void add(Object x);

  //@ req state<p>; ens result.ready;
  Iterator/*@<p,this>@*/ iterator();

}
```

A.3.2 The Iterator Interface

We repeat the `Iterator` interface from Section 2.5:

```
interface Iterator/*@<perm p, Collection iteratee>@*/ {

  //@ pred ready;                          // prestate for iteration cycle
  //@ pred readyForNext;                   // prestate for next()
  //@ pred readyForRemove<Object element>; // prestate for remove()

  //@ axiom ready -* iteratee.state<p>;    // stop iterating

  //@ req init * c.state<p> * c==iteratee;
  //@ ens ready;
  void init(Collection c);

  //@ req ready;
  //@ ens (result -* readyForNext) & (!result -* ready);
  boolean hasNext();

  //@ req readyForNext;
  //@ ens result.state<p> * readyForRemove<result>
  //@   * (result.state<p> * readyForRemove<result> -* ready);
```

`elem<p>`	datagroup: `state` associated with the node element `this.val` `p` is the permission for this datagroup
`left<p>`	datagroup: list prefix left of `this` (including element `states`) `p` is the permission for this datagroup
`right<p>`	datagroup: list postfix right of `this` (incl. element `states`) `p` is the permission for this datagroup
`succeeds<p,x>`	datagroup: entire list (including element `states`) `p` is the permission for this datagroup `x` is the previous node, `x != null` (Because `x != null`, `this` cannot be the header.)

`connects<p,x,y>`	datagroup: entire list (including element `states`)
	`p` is the permission for this datagroup
	`x` is the previous node, `x != null`
	`y` is the next node, `y != null`
	(Because `x`, `y != null`, `this` cannot the header or tailer.)

`succeedsBoth<p,x,xval,y>`	datagroup: entire list (incl. elem. `states`) except `xval.state`
	`p` is the permission for this datagroup
	`x` is the previous node, `x != null`
	`xval` is the element of the previous node
	`y` is the node two before `this` (i.e., previous to `x`), `y != null`
	(Because `x`, `y != null`, `this` or `x` cannot be the header.)

Figure 2: `Node` predicates

```
  Object next();

  //@ req readyForRemove<_> * p==1;
  //@ ens ready;
  void remove();

}
```

A.3.3 The Node Class

Nodes have three fields: `prev` points to the previous list node, `val` points to the node element, and `next` points to the next list node. We implement doubly-linked lists with header and tailer nodes (whose `val` fields are `null`). The header node is the only list node whose `prev` field is `null`; the tailer node is the only list node whose `next` field is `null`. The `Node` class defines several predicates. These are summarized informally in Figure 2.

```
final class Node {

  /*@ spec_public @*/ Node prev;
  /*@ spec_public @*/ Object val;
  /*@ spec_public @*/ Node next;

  //@ spec_public group elem<perm p> =
```

```
//@   (ex Node x)( val<p,x> * x.state<p> );

//@ spec_public group left<perm p> =
//@   (ex Node x)( prev<p,x> * x.next<p,this> * x.elem<p> * x.left<p>);

//@ spec_public group right<perm p> =
//@   (ex Node x)( next<p,x> * x.prev<p,this> * x.elem<p> * x.right<p> );

//@ spec_public extends group state<perm p> by
//@   left<p> * elem<p> * right<p>;

//@ spec_public group succeeds<perm p, Node x> =
//@   right<p> * elem<p> * prev<p,x> * x!=null *
//@   x.next<p,this> * x.elem<p> * x.left<p>;

//@ spec_public group connects<perm p, Node x, Node y> =
//@   prev<p,x> * x!=null * elem<p> * y!=null * next<p,y> *
//@   x.left<p> * x.elem<p> * x.next<p,this> *
//@   y.prev<p,this> * y.elem<p> * y.right<p>;

//@ spec_public group succeedsBoth<perm p, Node x, Object xval, Node y>
//@   right<p> * elem<p> * prev<p,x> * x!=null *
//@   x.next<p,this> * x.val<p,xval> * x.prev<p,y> * y!=null *
//@   y.next<p,this> * y.elem<p> * y.left<p>;

//@ axiom
//@   init -* state<1>;

//@ req val<p,x>;
//@ ens val<p,x> * x==result;
Object getVal() { return val; }

//@ req prev<p,x>;
//@ ens prev<p,x> * x==result;
Node getPrev() { return prev; }

//@ req next<p,x>;
//@ ens next<p,x> * x == result;
Node getNext() { return next; }

//@ req val<1,_>;
//@ ens val<1,x>;
void setVal(final Object x) { val = x; }

//@ req prev<1,_>;
//@ ens prev<1,x>;
void setPrev(final Node x) { prev = x; }

//@ req next<1,_>;
//@ ens next<1,x>;
```

```
  void setNext(final Node x) { next = x; }

}
```

A.3.4 The List Class

```
class List implements Collection {

  /*@ spec_public @*/ Node header;

  //@ spec_public extends group state<perm p> by
  //@   (ex Node x,y) ( header@List<p,x> * y.succeeds<p,x> * y!=null );

  //@ req init;
  //@ ens state<1>;
  public void init() {
    header = new Node();
    final Node tailer = new Node();
    tailer.setPrev(header);
    header.setNext(tailer);
  }

  //@ req state<p>;
  //@ ens result.ready;
  public Iterator/*@<p,this>@*/ iterator() {
    final Iterator/*@<p,this>@*/ iter = new ListIterator/*@<p,this>@*/();
    iter.init(this);
    return iter;
  }

  //@ req state<1> * x.state<1>;
  //@ ens state<1>;
  public void add(final Object x) {
    final Node newFirst = new Node();
    final Node oldFirst = header.getNext();
    oldFirst.setPrev(newFirst);
    newFirst.setNext(oldFirst);
    newFirst.setVal(x);
    newFirst.setPrev(header);
    header.setNext(newFirst);
  }

  //@ req state<p>;
  //@ ens (ex Node x)( header@List<p,x> * result.succeeds<p,x> *
  //@                  result!=null );
  Node getFirst() { return header.getNext(); }

}
```

Not only do we have to prove the method contracts, but also that the definition of the `state` datagroup is indeed a legal datagroup formula. Expressed more formally, we have to prove the following judgment (defined in Section H):

$\Gamma;\Gamma';\texttt{this} \vdash \texttt{(ex Node x,y)(this.header@List<p,x>} * y\texttt{.succeeds<p,x>)} : \texttt{grp}$
where $\Gamma = \texttt{this} : \texttt{List}$ and $\Gamma' = \texttt{p} : \texttt{perm}$

In particular, we have to prove that both existential quantifiers in this formula have a unique witness. Expressed more formally, for the inner existential we have to prove the following:

$\texttt{this} : \texttt{List}, \texttt{p} : \texttt{perm}, \texttt{x} : \texttt{Node}, \alpha : \texttt{Node}, \alpha' : \texttt{Node}; \texttt{this}; \texttt{true} \vdash'_{\mathsf{w}} F(\alpha)\ \texttt{\&}\ F(\alpha')\ \texttt{-*}\ \alpha\ \texttt{==}\ \alpha'$
where $F(\alpha) = \texttt{this.header@List<p,x>} * \alpha\texttt{.succeeds<p,x>}$

Here, $\vdash'_{\mathsf{w}}$ is the merge-restricted logical consequence judgment, as defined in Section G.

A.3.5 The List Iterator Class

```
final class ListIterator/*@<perm p, Collection iteratee>@*/
  implements Iterator/*@<p,iteratee>@*/
{

  Node current;

  //@ pred ready =
  //@   (ex Node x,y)(
  //@      PointsTo(current,1,x) * x!=null * x.succeeds<p,y> *
  //@      ( x.succeeds<p,y> -* iteratee.state<p> ) );

  //@ pred readyForNext =
  //@   (ex Node x,y,z)(
  //@      PointsTo(current,1,x) * x!=null * x.connects<p,y,z> *
  //@      ( x.succeeds<p,y> -* iteratee.state<p> ) );

  //@ pred readyForRemove<Object yval> =
  //@   (ex Node x,y,z)(
  //@      PointsTo(current,1,x) * x!=null * x.succeedsBoth<p,y,yval,z> *
  //@      ( yval.state<p> * x.succeedsBoth<p,y,yval,z> -* iteratee.state<p> )
  //@   );

  //@ req init * c.state<p> * c==iteratee;
  //@ ens ready;
  public void init(final Collection c) {
    current = ((List)c).getFirst();
  }

  //@ req ready;
  //@ ens (result -* readyForNext) & (!result -* ready);
  public boolean hasNext() {
    return current.getNext() != null;
```

```
  }

  //@ req readyForNext;
  //@ ens result.state<p> * readyForRemove<result> *
  //@   * (result.state<p> * readyForRemove<result> -* ready);
  public Object next() {
    final Object x = current.getVal();
    current = current.getNext();
    return x;
  }

  //@ req readyForRemove<_> * p==1;
  //@ ens ready;
  public void remove() {
    final Node newPrev = current.getPrev().getPrev();
    newPrev.setNext(current);
    current.setPrev(newPrev);
  }

}
```

B Notational Conventions and Derived Forms

We sometimes apply logical operators to sequences:

$$*() \triangleq \texttt{true} \quad *(F,\overline{G}) \triangleq F * *(\overline{G}) \quad \texttt{\&}() \triangleq \texttt{true} \quad \texttt{\&}(F,\overline{G}) \triangleq F \texttt{ \& \&}(\overline{G})$$

We sometimes use the following notation for sequences of formulas: If $F[\overline{e}]$ is a formula with an occurrence of a sequence $\overline{e}$ where a single e is expected, then $F[\overline{e}]$ is short for the formula sequence $(F[e_1],\ldots,F[e_n])$. If several sequences occur in the same formula, we implicitly assume that they all have the same length. For instance, when we write $F[\overline{e},\overline{v}]$, we implicitly assume $|\overline{e}| = |\overline{v}| = n$ and mean $(F[e_1,v_1],\ldots,F[e_n,v_n])$. We write $(qt\ T\ \overline{\alpha})(F)$ as a shorthand for $(qt\ T_1\ \alpha_1)(\ldots(qt\ T_n\ \alpha_n)(F))$, implicitly assuming that $|\overline{T}| = |\overline{\alpha}| = n$.

$e.\kappa\texttt{<}\overline{e}\texttt{>} \triangleq \texttt{(ex }\overline{T}\ \overline{\alpha}\texttt{)(*(}\overline{\alpha}\texttt{ == }\overline{e}\texttt{) * }e.\kappa\texttt{<}\overline{\alpha}\texttt{>)}$ where $\overline{T}$ are the types of $e.\kappa$'s parameters

$\texttt{(}qt\ t\texttt{<}\overline{e}\texttt{>}\ \alpha\texttt{)(}F\texttt{)} \triangleq \texttt{(ex }\overline{T}\ \overline{\alpha}'\texttt{)(*(}\overline{\alpha}'\texttt{ == }\overline{e}\texttt{) * (}qt\ t\texttt{<}\overline{\alpha}'\texttt{>}\ \alpha\texttt{)(}F\texttt{))}$
where $\overline{T}$ are the types of t's parameters

There is a similar derived form for commands:

If $\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ W\ m\texttt{(}\overline{V}\ \overline{\imath}\texttt{)}$ and $\overline{U} = \overline{T}[u/\texttt{this},\overline{v}/\overline{\imath}]$:
$\ell\texttt{=}u.m\texttt{<}\overline{e}\texttt{>(}\overline{v}\texttt{);}\ c \triangleq \texttt{unpack (ex }\overline{U}\ \overline{\alpha}\texttt{)(*(}\overline{\alpha}\texttt{ == }\overline{e}\texttt{));}\ \ell\texttt{=}u.m\texttt{<}\overline{\alpha}\texttt{>(}\overline{v}\texttt{);}\ c$

C Auxiliary Functions

Field Lookup, $\mathsf{fld}(C\texttt{<}\overline{\pi}\texttt{>}) = \overline{T}\ \overline{f}$**:**

(Fields Base)
$$\frac{}{\mathsf{fld}(\texttt{Object}) = \emptyset}$$

(Fields Ind)
$$\frac{\mathsf{fld}(D\texttt{<}\overline{\pi}'[\overline{\pi}/\overline{\alpha}]\texttt{>}) = \overline{F}' \qquad \textit{fin}\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ D\texttt{<}\overline{\pi}'\texttt{>}\ \texttt{impl}\ \overline{U}\ \texttt{\{}\overline{T}\ \overline{f}\ \textit{pd}\texttt{*}\ \textit{ax}\texttt{*}\ \textit{md}\texttt{*\}}}{\mathsf{fld}(C\texttt{<}\overline{\pi}\texttt{>}) = \overline{T}[\overline{\pi}/\overline{\alpha}]\ \overline{f},\overline{F}'}$$

Method Lookup, $\mathsf{mtype}(m,t\langle\overline{\pi}\rangle) = \mathit{fin}\ mt$ **and** $\mathsf{mbody}(m,C\langle\overline{\pi}\rangle) = \langle\overline{\alpha}\rangle(\overline{\imath}).c$**:**

(Mlkup Base)
$$\frac{\mathit{fin}\ \mathtt{class}\ C\langle\overline{T}'\ \overline{\alpha}'\rangle\ \mathtt{ext}\ U'\ \mathtt{impl}\ \overline{V}'\ \{\ldots\ \mathit{fin}\ \langle\overline{T}\ \overline{\alpha}\rangle\ \mathit{spec}\ U\ m(\overline{V}\ \overline{\imath})\ \{c\}\ \ldots\}}{\mathsf{mlkup}(m,C\langle\overline{\pi}\rangle) = (\mathit{fin}\ \langle\overline{T}\ \overline{\alpha}\rangle\ \mathit{spec}\ U\ m(\overline{V}\ \overline{\imath})\ \{c\})[\overline{\pi}/\overline{\alpha}']}$$

(Mlkup Ind) $m \notin \mathsf{dom}(md^*)$
$$\frac{\mathit{fin}\ \mathtt{class}\ C\langle\overline{T}\ \overline{\alpha}\rangle\ \mathtt{ext}\ D\langle\overline{\pi}'\rangle\ \mathtt{impl}\ \overline{U}\ \{fd^*\ pd^*\ ax^*\ md^*\} \quad \mathsf{mlkup}(m,D\langle\overline{\pi}'[\overline{\pi}/\overline{\alpha}]\rangle) = md'}{\mathsf{mlkup}(m,C\langle\overline{\pi}\rangle) = md'}$$

$$\mathsf{mbody}(m,C\langle\overline{\pi}\rangle) \triangleq \langle\overline{\alpha}\rangle(\mathtt{this},\overline{\imath}).c \quad \text{if } \mathsf{mlkup}(m,C\langle\overline{\pi}\rangle) = \mathit{fin}\ \langle\overline{T}\ \overline{\alpha}\rangle\ \mathit{spec}\ U\ m(\overline{V}\ \overline{\imath})\ \{c\}$$

For $m \notin \{\mathtt{fork},\mathtt{join}\}$:

$$\mathsf{mtype}(m,C\langle\overline{\pi}\rangle) \triangleq \mathit{fin}\ \langle\overline{T}\ \overline{\alpha}\rangle\ \mathtt{req}\ F;\mathtt{ens}\ (\mathtt{ex}\ U\ \mathtt{result})\ (G);\ U\ m(C\langle\overline{\pi}\rangle\ \mathtt{this},\overline{V}\ \overline{\imath})$$
$$\text{if } \mathsf{mlkup}(m,C\langle\overline{\pi}\rangle) = \mathit{fin}\ \langle\overline{T}\ \overline{\alpha}\rangle\ \mathtt{req}\ F;\mathtt{ens}\ G;\ U\ m(\overline{V}\ \overline{\imath})\ \{c\}$$

For $m \in \{\mathtt{fork},\mathtt{join}\}$:

$$\mathsf{mtype}(\mathtt{fork},C\langle\overline{\pi}\rangle) \triangleq \mathtt{final}\ \mathtt{req}\ F;\mathtt{ens}\ (\mathtt{ex}\ \mathtt{void}\ \alpha)\ (\mathtt{true});\ \mathtt{void}\ \mathtt{fork}(C\langle\overline{\pi}\rangle\ \iota)$$
$$\text{if } \mathsf{mtype}(\mathtt{run},C\langle\overline{\pi}\rangle) = \mathit{fin}\ \mathtt{req}\ F;\mathtt{ens}\ G;\ \mathtt{void}\ \mathtt{run}(C\langle\overline{\pi}\rangle\ \iota)$$
$$\mathsf{mtype}(\mathtt{join},C\langle\overline{\pi}\rangle) \triangleq \mathtt{final}\ \mathtt{req}\ \mathtt{true};\mathtt{ens}\ G;\ \mathtt{void}\ \mathtt{join}(C\langle\overline{\pi}\rangle\ \iota)$$
$$\text{if } \mathsf{mtype}(\mathtt{run},C\langle\overline{\pi}\rangle) = \mathit{fin}\ \mathtt{req}\ F;\mathtt{ens}\ G;\ \mathtt{void}\ \mathtt{run}(C\langle\overline{\pi}\rangle\ \iota)$$

(Mtype Interface)
$$\frac{\mathtt{interface}\ I\langle\overline{T}\ \overline{\alpha}\rangle\ \mathtt{ext}\ \overline{U}\ \{\ldots\langle\overline{T}'\ \overline{\alpha}'\rangle\ \mathtt{req}\ F;\mathtt{ens}\ G;\ U'\ m(\overline{V}'\ \overline{\imath})\ \ldots\}}{\mathsf{mtype}(m,I\langle\overline{\pi}\rangle) = (\langle\overline{T}'\ \overline{\alpha}'\rangle\mathtt{req}\ F;\mathtt{ens}\ (\mathtt{ex}\ U'\ \mathtt{result})\ (G);U'\ m(I\langle\overline{\pi}\rangle\mathtt{this},\overline{V}'\ \overline{\imath}))[\overline{\pi}/\overline{\alpha}]}$$

In (Mtype Interface), note that we do not model Java's "inheritance of method signatures", but instead require that each method signature is repeated in interfaces. This is not a significant restriction, because inherited method signatures can be filled in at compile time.

Predicate Lookup, $\mathsf{ptype}(P,t\langle\overline{\pi}\rangle) = \mathit{fin}\ pt$ **and** $\mathsf{pbody}(\pi.P\langle\overline{\pi}'\rangle,C\langle\overline{\pi}''\rangle) = F\ \mathsf{ext}\ T$**:**

$\mathsf{plkup}(\mathtt{init},\mathtt{Object}) = \mathtt{pred}\ \mathtt{init}=\mathtt{true}\ \mathsf{ext}\ \mathtt{Object}$
$\mathsf{plkup}(\mathtt{init},\mathtt{Thread}) = \mathtt{pred}\ \mathtt{init}=\mathtt{Perm}(\mathtt{this}[\mathtt{join}],1)\ \mathsf{ext}\ \mathtt{Object}$
$\mathsf{plkup}(\mathtt{state},\mathtt{Object}) = \mathtt{group}\ \mathtt{state}\langle\mathtt{perm}\ \alpha\rangle=\mathtt{true}\ \mathsf{ext}\ \mathtt{Object}$

(Plkup Base) $\mathsf{plkup}(P,U) = \mathsf{undef}$
$$\frac{\mathit{fin}\ \mathtt{class}\ C\langle\overline{T}'\ \overline{\alpha}'\rangle\ \mathtt{ext}\ U\ \mathtt{impl}\ \overline{V}\ \{\ldots\ \mathit{fin}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle=F\ \ldots\}}{\mathsf{plkup}(P,C\langle\overline{\pi}\rangle) = (\mathit{fin}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle=F\ \mathsf{ext}\ \mathtt{Object})[\overline{\pi}/\overline{\alpha}']}$$

(Plkup Extend) $P \neq \mathtt{init}$
$$\frac{\mathit{fin}\ \mathtt{class}\ C\langle\overline{T}'\ \overline{\alpha}'\rangle\ \mathtt{ext}\ U\ \mathtt{impl}\ \overline{V}\ \{\ldots\ \mathit{fin}\ \mathtt{ext}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle\ \mathtt{by}\ F\ \ldots\}}{\mathsf{plkup}(P,C\langle\overline{\pi}\rangle) = (\mathit{fin}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle=F\ \mathsf{ext}\ U)[\overline{\pi}/\overline{\alpha}']}$$

(Plkup Inherit) $P \notin \mathsf{dom}(pd^*) \quad \mathsf{plkup}(P,U) = \mathit{fin}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle=F\ \mathsf{ext}\ U'$
$$\frac{\mathit{fin}\ \mathtt{class}\ C\langle\overline{T}'\ \overline{\alpha}'\rangle\ \mathtt{ext}\ U\ \mathtt{impl}\ \overline{V}\ \{fd^*\ pd^*\ ax^*\ md^*\}}{\mathsf{plkup}(P,C\langle\overline{\pi}\rangle) = (\mathit{fin}\ \mathit{pmod}\ P\langle\overline{T}\ \overline{\alpha}\rangle=\mathtt{true}\ \mathsf{ext}\ U)[\overline{\pi}/\overline{\alpha}']}$$

(Plkup `init`) $C \neq \mathtt{Thread}$
$$\frac{\mathit{fin}\ \mathtt{class}\ C\langle\overline{T}'\ \overline{\alpha}'\rangle\ \mathtt{ext}\ U\ \mathtt{impl}\ \overline{V}\ \{\overline{T}\ \overline{f}\ pd^*\ ax^*\ md^*\}}{\mathsf{plkup}(\mathtt{init},C\langle\overline{\pi}\rangle) = (\ \mathtt{pred}\ \mathtt{init}=\mathtt{PointsTo}(\mathtt{this}[\overline{f}],1,\mathsf{df}(\overline{T}))\ \mathsf{ext}\ U\ )[\overline{\pi}/\overline{\alpha}']}$$

$$\mathsf{pbody}(\pi.P\texttt{<}\overline{\pi'}\texttt{>},U) \triangleq (F\ \mathsf{ext}\ V)[\pi/\texttt{this},\overline{\pi'}/\overline{\alpha}] \quad \text{if } \mathsf{plkup}(P,U) = \mathit{fin\ pmod}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} = F\,\mathsf{ext}\,V$$
$$\mathsf{ptype}(P,C\texttt{<}\overline{\pi}\texttt{>}) \triangleq \mathit{fin\ pmod}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} \quad \text{if } \mathsf{plkup}(P,C\texttt{<}\overline{\pi}\texttt{>}) = \mathit{fin\ pmod}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} = F\,\mathsf{ext}\,V$$

(Ptype Interface)
$$\frac{\texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ \overline{U}\ \texttt{\{}\ldots\ \mathit{pmod}\ P\texttt{<}\overline{T'}\ \overline{\alpha'}\texttt{>}\ldots\texttt{\}} \quad P \neq \texttt{init}}{\mathsf{ptype}(P,I\texttt{<}\overline{\pi}\texttt{>}) = (\mathit{pmod}\ P\texttt{<}\overline{T'}\ \overline{\alpha'}\texttt{>})[\overline{\pi}/\overline{\alpha}]}$$

(Ptype Interface Implicit)
$$\frac{\texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ \overline{U}\ \texttt{\{}\mathit{pt}\texttt{*}\ \mathit{ax}\texttt{*}\ \mathit{mt}\texttt{*\}} \quad P \in \{\texttt{state},\texttt{init}\} \quad P \notin \mathsf{dom}(\mathit{pt}\texttt{*})}{\mathsf{ptype}(P,I\texttt{<}\overline{\pi}\texttt{>}) = \mathsf{ptype}(P,\texttt{Object})}$$

$$\mathsf{arity}(P,C) \triangleq n \quad \text{if } (\exists\overline{\pi})(\mathsf{ptype}(P,C\texttt{<}\overline{\pi}\texttt{>}) = \mathit{fin\ pmod}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} \text{ and } |\overline{\alpha}| = n)$$

Axiom Lookup, $\mathsf{axiom}(t\texttt{<}\overline{\pi}\texttt{>}) = F$**:**

$$\mathsf{axiom}(\mathit{ax}\texttt{*}) \triangleq \begin{cases} \texttt{true} & \text{if } \mathit{ax}\texttt{*} = () \\ F * \mathsf{axiom}(\mathit{ax}\texttt{*}) & \text{if } \mathit{ax}\texttt{*} = (\texttt{axiom}\ F, \mathit{ax}\texttt{*}) \end{cases}$$
$$\mathsf{axiom}(\overline{T}) \triangleq \begin{cases} \texttt{true} & \text{if } \overline{T} = () \text{ or } \overline{T} = (\texttt{Object}) \\ \mathsf{axiom}(U) * \mathsf{axiom}(\overline{V}) & \text{if } \overline{T} = (U,\overline{V}) \end{cases}$$

(Ax Class)
$$\frac{\mathit{fin}\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \overline{V}\ \texttt{\{}\mathit{fd}\texttt{*}\ \mathit{pd}\texttt{*}\ \mathit{ax}\texttt{*}\ \mathit{md}\texttt{*\}}}{\mathsf{axiom}(C\texttt{<}\overline{\pi}\texttt{>}) = \mathsf{axiom}(\mathit{ax}\texttt{*}[\overline{\pi}/\overline{\alpha}]) * \mathsf{axiom}((U,\overline{V})[\overline{\pi}/\overline{\alpha}])}$$

(Ax Interface)
$$\frac{\texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ \overline{U}\ \texttt{\{}\mathit{pt}\texttt{*}\ \mathit{ax}\texttt{*}\ \mathit{mt}\texttt{*\}}}{\mathsf{axiom}(I\texttt{<}\overline{\pi}\texttt{>}) = \mathsf{axiom}(\mathit{ax}\texttt{*}[\overline{\pi}/\overline{\alpha}]) * \mathsf{axiom}(\overline{U}[\overline{\pi}/\overline{\alpha}])}$$

D Typing Rules

D.1 Operator Types and Semantics

Let arity be a function that assigns to each operator its arity. We assume:

$$\mathsf{arity}(\texttt{==}) \triangleq \mathsf{arity}(\texttt{\&}) \triangleq \mathsf{arity}(\texttt{|}) \triangleq 2 \qquad \mathsf{arity}(\texttt{!}) \triangleq \mathsf{arity}(C\ \texttt{isclassof}) \triangleq 1$$

Let type be a function that maps each operator *op* to a partial function $\mathsf{type}(\mathit{op})$ of type $\{\texttt{int},\texttt{bool},\texttt{Object},\texttt{perm}\}^{\mathsf{arity}(\mathit{op})} \rightharpoonup \{\texttt{int},\texttt{bool},\texttt{perm}\}$. We assume:

$$\mathsf{type}(\texttt{==}) \triangleq \{\ ((T,T),\texttt{bool})\ \mid\ T \in \{\texttt{int},\texttt{bool},\texttt{Object},\texttt{perm}\}\ \} \quad \mathsf{type}(\texttt{!}) \triangleq \{\ (\texttt{bool},\texttt{bool})\ \}$$
$$\mathsf{type}(C\ \texttt{isclassof}) \triangleq \{\ (\texttt{Object},\texttt{bool})\ \} \quad \mathsf{type}(\texttt{\&}) \triangleq \mathsf{type}(\texttt{|}) \triangleq \{\ ((\texttt{bool},\texttt{bool}),\texttt{bool})\ \}$$

Let $[\![\texttt{bool}]\!]^h = \{\texttt{true},\texttt{false}\}$, $[\![\texttt{int}]\!]^h = \mathsf{Int}$ and $[\![\texttt{Object}]\!]^h = \mathsf{dom}(h)$ and $[\![\texttt{perm}]\!]^h = \{\texttt{split}^n(1) \mid n \geq 0\}$ and $[\![T_1,\ldots,T_n]\!]^h = [\![T_1]\!]^h \times \cdots \times [\![T_n]\!]^h$. We assume that $[\![\mathit{op}]\!]^h$ is a function of the following type:

$$[\![\mathit{op}]\!]^h \ \in\ \bigcup_{(\overline{T},U)\in\mathsf{type}(\mathit{op})} [\![\overline{T}]\!]^h \to [\![U]\!]^h$$

For the logical operators `!`, `|` and `&`, we assume the usual interpretations. `==` is interpreted as the identity relation. $[\![C\ \texttt{isclassof}]\!]^h(o) = \texttt{true}$ whenever $h(o)_1 = C\texttt{<}\overline{\pi}\texttt{>}$ for some $\overline{\pi}$, otherwise $[\![C\ \texttt{isclassof}]\!]^h(o) = \texttt{false}$.

D.2 Type Environments and Types

A *type environment* is a partial function of type $\mathsf{ObjId} \cup \mathsf{Var} \rightharpoonup \mathsf{Ty}$. We use the meta-variable Γ to range over type environments. Γ_{hp} denotes the *restriction of* Γ *to* ObjId:

$$\Gamma_{\mathsf{hp}} \triangleq \{ (o,T) \in \Gamma \mid o \in \mathsf{ObjId} \}$$

We define a *heap extension order* on well-formed type environments:

$$\Gamma' \supseteq_{\mathsf{hp}} \Gamma \quad \text{iff} \quad \Gamma' \vdash \diamond,\ \Gamma \vdash \diamond,\ \Gamma' \supseteq \Gamma \text{ and } \Gamma'_{|\mathsf{Var}} = \Gamma_{|\mathsf{Var}}$$

Good Environments, $\Gamma \vdash \diamond$:

$$\text{(Env)}\quad \frac{(\forall x \in \mathsf{dom}(\Gamma))(\Gamma \vdash \Gamma(x) : \diamond) \quad (\forall o \in \mathsf{dom}(\Gamma))(\Gamma_{\mathsf{hp}} \vdash \Gamma(o) : \diamond) \quad (\forall \alpha^{\mathsf{val}} \in \mathsf{dom}(\Gamma))(\Gamma(\alpha^{\mathsf{val}}) \neq \mathtt{perm}) \quad (\forall \alpha^{\mathsf{perm}} \in \mathsf{dom}(\Gamma))(\Gamma(\alpha^{\mathsf{perm}}) = \mathtt{perm})}{\Gamma \vdash \diamond}$$

Note that types assigned to object ids cannot have free variables:

Lemma 1 *If* $(\Gamma \vdash \diamond)$, *then* $(\Gamma_{\mathsf{hp}} \vdash \diamond)$.

Good Types, $\Gamma \vdash T : \diamond$:

$$\text{(Ty Void)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathtt{void} : \diamond} \qquad \text{(Ty Int)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathtt{int} : \diamond} \qquad \text{(Ty Bool)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathtt{bool} : \diamond}$$

$$\text{(Ty Ref)}\ \frac{t\langle \overline{T}\ \overline{\alpha} \rangle \in ct \quad \Gamma \vdash \diamond \quad \Gamma \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}]}{\Gamma \vdash t\langle \overline{\pi} \rangle : \diamond} \qquad \text{(Ty Perm)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathtt{perm} : \diamond}$$

D.3 Values, Expressions, Formulas

Well-typed Values and Expressions, $\Gamma \vdash v : T$, $\Gamma \vdash \pi : T$ and $\Gamma \vdash e : T$:

$$\text{(Val Null)}\ \frac{\Gamma \vdash t\langle \overline{\pi} \rangle : \diamond}{\Gamma \vdash \mathtt{null} : t\langle \overline{\pi} \rangle} \qquad \text{(Val Int)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \mathtt{int}} \qquad \text{(Val Bool)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash b : \mathtt{bool}}$$

$$\text{(Val Id)}\ \frac{\Gamma \vdash \diamond \quad \Gamma(v) = T}{\Gamma \vdash v : T} \qquad \text{(Val/Exp Sub)}\ \frac{\Gamma \vdash e : T \quad T <: U}{\Gamma \vdash e : U}$$

$$\text{(Exp Get)}\ \frac{\Gamma \vdash e : C\langle \overline{\pi} \rangle \quad T\ f \in \mathsf{fld}(C\langle \overline{\pi} \rangle)}{\Gamma \vdash e.f : T} \qquad \text{(Exp Op)}\ \frac{\Gamma \vdash \overline{e} : \overline{U} \quad \mathsf{type}(op)(\overline{U}) = T}{\Gamma \vdash op(\overline{e}) : T}$$

$$\text{(Exp Full)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash 1 : \mathtt{perm}} \qquad \text{(Exp Split)}\ \frac{\Gamma \vdash e : \mathtt{perm}}{\Gamma \vdash \mathtt{split}(e) : \mathtt{perm}}$$

We extend the partial function $\mathsf{ptype}(P, t\langle \overline{\pi} \rangle)$ to predicate selectors:

$$\mathsf{ptype}(P@C, t\langle \overline{\pi} \rangle) \triangleq \begin{cases} \mathsf{ptype}(P, t\langle \overline{\pi} \rangle) & \text{if } t = C \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Well-typed Formulas, $\Gamma \vdash F : \diamond$:

$$\text{(Form Bool)}\ \frac{\Gamma \vdash e : \mathtt{bool}}{\Gamma \vdash e : \diamond} \qquad \text{(Form Points To)}\ \frac{\Gamma \vdash \pi : \mathtt{perm} \quad \Gamma \vdash e : U \quad T\ f \in \mathsf{fld}(U) \quad \Gamma \vdash e' : T}{\Gamma \vdash \mathtt{PointsTo}(e\mathtt{[}f\mathtt{]}, \pi, e') : \diamond}$$

$$\text{(Form Perm)}\ \frac{\Gamma \vdash \pi : \mathtt{perm} \quad \Gamma \vdash e : \mathtt{Thread}}{\Gamma \vdash \mathtt{Perm}(e\mathtt{[join]}, \pi) : \diamond}$$

$$\text{(Form Pure)}\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \mathtt{Pure}(e) : \diamond} \qquad \text{(Form Pred)}\ \frac{\Gamma \vdash \pi : U \quad \mathsf{ptype}(\kappa, U) = \mathit{fin}\ \mathit{pmod}\ P\langle \overline{T}\ \overline{\alpha} \rangle \quad \Gamma \vdash \overline{\pi'} : \overline{T}}{\Gamma \vdash \pi.\kappa\langle \overline{\pi'} \rangle : \diamond}$$

$$\text{(Form Log Op)}\ \frac{\Gamma \vdash F : \diamond \quad \Gamma \vdash F' : \diamond}{\Gamma \vdash F\ lop\ F' : \diamond} \qquad \text{(Form Quant)}\ \frac{\Gamma \vdash T : \diamond \quad \Gamma, \alpha : T \vdash F : \diamond}{\Gamma \vdash (qt\ T\ \alpha)\,(F) : \diamond}$$

D.4 Runtime Structures

Well-typed Objects, $\Gamma \vdash obj : \diamond$:

$$\text{(Obj)}\ \frac{\mathsf{dom}(os) \subseteq \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>})) \quad \Gamma \vdash C\texttt{<}\bar{\pi}\texttt{>} : \diamond \quad (\forall f \in \mathsf{dom}(os))(T\ f \in \mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}) \Rightarrow \Gamma \vdash os(f) : T)}{\Gamma \vdash (C\texttt{<}\bar{\pi}\texttt{>}, os) : \diamond}$$

Note that we require $\mathsf{dom}(os) \subseteq \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))$, not $\mathsf{dom}(os) = \mathsf{dom}(\mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>}))$. Thus, we allow partial objects. This is needed, because our semantics of $*$ splits heaps on a per-field basis.

Well-typed Heaps and Stacks, $\Gamma \vdash h : \diamond$ and $\Gamma \vdash s : \diamond$:

$$\text{(Heap)}\ \frac{\Gamma \vdash \diamond \quad \Gamma = \mathsf{fst} \circ h \quad (\forall o \in \mathsf{dom}(h))(\Gamma \vdash h(o) : \diamond)}{\Gamma \vdash h : \diamond} \qquad \text{(Stack)}\ \frac{\Gamma \vdash \diamond \quad (\forall x \in \mathsf{dom}(s))(\Gamma \vdash s(x) : \Gamma(x))}{\Gamma \vdash s : \diamond}$$

Note that the heap typing judgment does not satisfy weakening, as $(\Gamma \vdash h : \diamond)$ implies $\mathsf{dom}(h) = \mathsf{dom}(\Gamma)$. This is intentional.

E Operational Semantics

We define functions that map types to their default values, and object types to their initial object stores:

$$\mathsf{df} : \mathsf{Ty} \to \mathsf{ClVal} \quad \mathsf{df}(C\texttt{<}\pi\texttt{>}) \triangleq \texttt{null} \quad \mathsf{df}(\texttt{void}) \triangleq \texttt{null} \quad \mathsf{df}(\texttt{int}) \triangleq 0 \quad \mathsf{df}(\texttt{bool}) \triangleq \texttt{false}$$
$$\mathsf{init} : \mathsf{Ty} \rightharpoonup \mathsf{ObjStore} \quad \mathsf{init}(C\texttt{<}\pi\texttt{>})(f) \triangleq \mathsf{df}(T), \text{ if } (T\ f) \in \mathsf{fld}(C\texttt{<}\pi\texttt{>})$$

We extend the syntax by a `return` command.

$$c ::= \ldots \mid \ell\texttt{=return(}v\texttt{);}\, c \mid \ldots$$

This command is not allowed to be used in source programs. Operationally, it is a no-op. It is used as a syntactic marker for the "points" where the receiver changes. It is associated with a special Hoare rule. Because the `return` command does not occur in source programs, this Hoare rule is never used in actual program verifications. Its only intent is to help us state a smooth global invariant.

$$\text{(Return)}\ \frac{\Gamma \vdash v : T \quad \Gamma; o; F \vdash G[v/\alpha] \quad T <: U \quad \Gamma, \ell : U; p \vdash \{(\texttt{ex}\ T\ \alpha)\,(\alpha\ \texttt{==}\ \ell\ *\ G)\}c : V\{H\}}{\Gamma, \ell : U; o \vdash \{F\}\ell\texttt{=return(}v\texttt{);}\, c : V\{H\}}$$

For the operational semantics of method calls, we define a derived form, $\ell \leftarrow c; c'$, which assigns the result of a computation c to variable ℓ. In our applications of this derived form, its argument c is always a source program command and we therefore assume that c does not contain `return` commands.

$$
\begin{array}{rcll}
\ell \leftarrow v; c & \triangleq & \ell\texttt{=return(}v\texttt{)}; c & \\
\ell \leftarrow (U\ \ell'; c); c' & \triangleq & U\ \ell'; \ell \leftarrow c; c' & \text{if } \ell' \notin \mathsf{fv}(c'),\ \ell' \neq \ell \\
\ell \leftarrow (\texttt{final}\ U\ \imath\texttt{=}\ell'; c); c' & \triangleq & \texttt{final}\ U\ \imath\texttt{=}\ell'; \ell \leftarrow c; c' & \text{if } \imath \notin \mathsf{fv}(c') \\
\ell \leftarrow (\texttt{unpack (ex}\ T\ \alpha\texttt{)(}F\texttt{)}; c); c' & \triangleq & \texttt{unpack (ex}\ T\ \alpha\texttt{)(}F\texttt{)}; \ell \leftarrow c; c' & \text{if } \alpha \notin \mathsf{fv}(c') \\
\ell \leftarrow (hc; c); c' & \triangleq & hc; \ell \leftarrow c; c' &
\end{array}
$$

Furthermore, we define sequential composition of commands:

$$c; c' \ \triangleq\ \texttt{void}\ \ell; \ell \leftarrow c; c' \quad \text{where } \ell \notin \mathsf{fv}(c, c')$$

We use the following abbreviation for field updates:

$$h[o.f \mapsto v] \ \triangleq\ h[o \mapsto (h(o)_1, h(o)_2[f \mapsto v])]$$

The state reduction relation $\rightarrow_{ct}$ is given with respect to a class table ct. We follow the usual convention to omit the subscript ct unless we want to emphasize its existence.

State Reductions, $st \rightarrow_{ct} st'$**:**

(Red Dcl) $\ell \notin \mathsf{dom}(s) \quad s' = s[\ell \mapsto \mathsf{df}(T)]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ T\ \ell; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red Fin Dcl) $s(\ell) = v \quad c' = c[v/\imath]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{final}\ T\ \imath\texttt{=}\ell; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c')\rangle$

(Red Unpack)
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{unpack (ex}\ T\ \alpha\texttt{)(}F\texttt{)}; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c)\rangle$

(Red Var Set) $s' = s[\ell \mapsto v]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}v; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red Op) $\mathsf{arity}(op) = |\overline{v}| \quad [\![op]\!]^h(\overline{v}) = w \quad s' = s[\ell \mapsto w]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}op(\overline{v}); c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red Get) $s' = s[\ell \mapsto h(o)_2(f)]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}o.f; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red Set) $h' = h[o.f \mapsto v]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ fin\ o.f\texttt{=}v; c)\rangle \rightarrow \langle h', ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c)\rangle$

(Red Cast) $h(v)_1 <: T \quad s' = s[\ell \mapsto v]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=(}T\texttt{)}v; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red New) $o \notin \mathsf{dom}(h) \quad h' = h[o \mapsto (C\texttt{<}\overline{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\overline{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto o]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=new}\ C\texttt{<}\overline{\pi}\texttt{>}; c)\rangle \rightarrow \langle h', ts \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ c)\rangle$

(Red If True)
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{if (true)\{}c\texttt{\}else\{}c'\texttt{\}}; c'')\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c; c'')\rangle$

(Red If False)
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{if (false)\{}c\texttt{\}else\{}c'\texttt{\}}; c'')\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ c'; c'')\rangle$

(Red Call) $m \notin \{\texttt{fork}, \texttt{join}\}$
$h(o)_1 = C\texttt{<}\overline{\pi}'\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\overline{\pi}'\texttt{>}) = \texttt{<}\overline{\alpha}, \overline{\alpha}'\texttt{>}(\imath_0, \overline{\imath}).c_m \quad c' = c_m[\overline{\pi}/\overline{\alpha}, o/\imath_0, \overline{v}/\overline{\imath}]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}o.m\texttt{<}\overline{\pi}\texttt{>(}\overline{v}\texttt{)}; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell \leftarrow c'; c)\rangle$

(Red Return)
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=return(}v\texttt{)}; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}v; c)\rangle$

(Red Fork) $h(o)_1 = C\texttt{<}\overline{\pi}\texttt{>} \quad o \notin \mathsf{dom}(ts), \{p\} \quad \mathsf{mbody}(\texttt{run}, C\texttt{<}\overline{\pi}\texttt{>}) = \texttt{<>}(\imath).c_r \quad c_o = c_r[o/\imath]$
$\langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}o.\texttt{fork()}; c)\rangle \rightarrow \langle h, ts \mid p\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=null}; c) \mid o\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c_o)\rangle$

(Red Join)
$\langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=}o\texttt{.join();}\ c) \mid o \text{ is } (s' \text{ in } v)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } \ell\texttt{=null;}\ c) \mid o \text{ is } (s' \text{ in } v)\rangle$
(Red Assert)
$\langle h, ts \mid p \text{ is } (s \text{ in } \texttt{assert(}F\texttt{);}\ c)\rangle \rightarrow \langle h, ts \mid p \text{ is } (s \text{ in } c)\rangle$

Note that, in (Red Call), the method body may have more logic parameters than the caller supplies. This is because our notion of method subtyping allows to increase the number of logic parameters in subtypes.

F Natural Deduction Rules

Logical Consequence, $\Gamma; v; \bar{F} \vdash G$**:**

(Id) $\dfrac{\Gamma; v \vdash \bar{F}, G : \diamond}{\Gamma; v; \bar{F}, G \vdash G}$ (Ax) $\dfrac{\Gamma; v \vdash G \quad \Gamma; v \vdash \bar{F}, G : \diamond \quad \bar{F} \vdash G : \checkmark}{\Gamma; v; \bar{F} \vdash G}$ (Pure Intro) $\dfrac{\Gamma; v; \bar{F} \vdash G \quad \bar{F} \vdash e : \checkmark \quad \Gamma \vdash e : T}{\Gamma; v; \bar{F} \vdash G * \texttt{Pure}(e)}$

($*$ Intro) $\dfrac{\Gamma; v; \bar{F} \vdash H_1 \quad \Gamma; v; \bar{G} \vdash H_2}{\Gamma; v; \bar{F}, \bar{G} \vdash H_1 * H_2}$ ($*$ Elim) $\dfrac{\Gamma; v; \bar{F} \vdash G_1 * G_2 \quad \Gamma; v; \bar{E}, G_1, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$

($-\!\!*$ Intro) $\dfrac{\Gamma; v; \bar{F}, G_1 \vdash G_2 \quad \bar{F} \vdash G_1 : \checkmark}{\Gamma; v; \bar{F} \vdash G_1 -\!\!* G_2}$ ($-\!\!*$ Elim) $\dfrac{\Gamma; v; \bar{F} \vdash H_1 -\!\!* H_2 \quad \Gamma; v; \bar{G} \vdash H_1}{\Gamma; v; \bar{F}, \bar{G} \vdash H_2}$

($\&$ Intro) $\dfrac{\Gamma; v; \bar{F} \vdash G_1 \quad \Gamma; v; \bar{F} \vdash G_2}{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}$ ($\&$ Elim 1) $\dfrac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_1}$ ($\&$ Elim 2) $\dfrac{\Gamma; v; \bar{F} \vdash G_1 \,\&\, G_2}{\Gamma; v; \bar{F} \vdash G_2}$

($|$ Intro 1) $\dfrac{\Gamma; v; \bar{F} \vdash G_1 \quad \bar{F} \vdash G_2 : \checkmark}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}$ ($|$ Intro 2) $\dfrac{\Gamma; v; \bar{F} \vdash G_2 \quad \bar{F} \vdash G_1 : \checkmark}{\Gamma; v; \bar{F} \vdash G_1 \mid G_2}$ ($|$ Elim) $\dfrac{\Gamma; v; \bar{F} \vdash G_1 \mid G_2 \quad \Gamma; v; \bar{E}, G_1 \vdash H \quad \Gamma; v; \bar{E}, G_2 \vdash H}{\Gamma; v; \bar{F}, \bar{E} \vdash H}$

(Ex Intro) $\dfrac{\Gamma \vdash \pi : T \quad \Gamma; v; \bar{F} \vdash G[\pi/\alpha]}{\Gamma; v; \bar{F} \vdash (\texttt{ex}\ T\ \alpha)(G)}$ (Ex Elim) $\dfrac{\alpha \notin \bar{F}, H \quad \Gamma; v; \bar{E} \vdash (\texttt{ex}\ T\ \alpha)(G) \quad \Gamma, \alpha : T; v; \bar{F}, G \vdash H}{\Gamma; v; \bar{E}, \bar{F} \vdash H}$

(Fa Intro) $\dfrac{\alpha \notin \bar{F} \quad \Gamma, \alpha : T; v; \bar{F} \vdash G}{\Gamma; v; \bar{F} \vdash (\texttt{fa}\ T\ \alpha)(G)}$ (Fa Elim) $\dfrac{\Gamma; v; \bar{F} \vdash (\texttt{fa}\ T\ \alpha)(G) \quad \Gamma \vdash \pi : T}{\Gamma; v; \bar{F} \vdash G[\pi/\alpha]}$

G Supported Formulas

We now explain the premise $\Gamma; v \vdash F : \texttt{supp}$ in the split/merge rule. We note first that, if we did not have abstract predicates, split/merging atomic predicates (i.e, `PointsTo`, `Pure` and boolean expressions) would be good enough. But if we want to split whole datagroups (which are defined by composite formulas), we need to be able to split composite formulas. It is easy to show the soundness of split/merge for all atomic predicates, $*$ and $\&$. For the split-direction (left-to-right) only the soundness proof for linear

implication is problematic. For the merge direction, the cases for disjunction and existentials are problematic, too. Unfortunately, almost all typical datagroups are defined by formulas that use existentials in the disguise of the derived form $\texttt{Perm}(e\texttt{[}f\texttt{]},\pi)$. Fortunately, the merge axioms can be proven sound if existentials in $\mathsf{split}(F)$ have unique witnesses. We say that $(\texttt{ex}\ T\ \alpha)(F)$ has a *unique witness* if the validity of $F[\pi/\alpha]$ and $F[\pi'/\alpha]$ implies $\pi == \pi'$. Some simple examples:

- $(\texttt{ex}\ T\ \alpha)(\texttt{PointsTo}(o\texttt{[}f\texttt{]},\pi,\alpha))$: This existential has a unique witness, because in every model $o.f$ points to at most one value.
- $(\texttt{ex}\ T\ \alpha)(\texttt{PointsTo}(\alpha\texttt{[}f\texttt{]},\pi,o))$: This existential does not have a unique witness, because there are models where more than one pointer to o exists.
- $(\texttt{ex perm}\ \alpha)(\texttt{PointsTo}(o\texttt{[}f\texttt{]},\alpha,p))$: This existential does not have a unique witness. In all models where $\mathscr{P}(o,f)=1$ and $h(o)_2(f)=p$ both 1 and $\texttt{split}(1)$ are witnesses.

Informally, we define supported formulas as formulas F that do not contain `-*`, `|`, `fa`, all predicate identifiers in F are datagroup identifiers, and all existentials in F have unique witnesses. Formally, we define supportedness proof-theoretically:

We first define a variant of logical consequence that restricts the merge axiom. In the following, *let F^{sp} range over formulas that do not contain* `-*`, `|`, `fa` *or predicate identifiers that are not data group identifiers (but may contain existentials).*

Weakly Merge-restricted Logical Consequence, $\Gamma;v;\overline{F}\vdash'_{\mathsf{w}}\ G$**:**

The proof rules are the same as for $\Gamma;v;\overline{F}\vdash G$, except that we replace the split/merge axiom by the following axioms:

$\Gamma;v\vdash'_{\mathsf{w}}\ F^{\mathsf{sp}}\ \texttt{-*}\ (\mathsf{split}(F^{\mathsf{sp}})*\mathsf{split}(F^{\mathsf{sp}}))$

F^{sp} does not contain datagroup ids or existentials $\Rightarrow\ \Gamma;v\vdash'_{\mathsf{w}}\ (\mathsf{split}(F^{\mathsf{sp}})*\mathsf{split}(F^{\mathsf{sp}}))\ \texttt{-*}\ F^{\mathsf{sp}}$

We also define a *strongly merge-restricted logical consequence judgment* $\vdash'$:

$$\Gamma;v;\overline{F}\vdash' G \quad \text{iff} \quad \begin{cases}\text{there is a proof of } \Gamma;v;\overline{F}\vdash'_{\mathsf{w}} G \text{ that does not use}\\ \text{class axioms or opening/closing of abstract predicates}\end{cases}$$

Now, we define the judgment $(\Gamma;v\vdash F:\mathsf{supp})$. The only interesting rule is (Supp Ex), whose last premise enforces unique existential witnesses. All other rules are instances of the well-typedness rules for formulas, restricted to operators that are unproblematic for split/merge.

Supported Formulas, $\Gamma;v\vdash F:\mathsf{supp}$**:**

$$\frac{\Gamma\vdash e:\texttt{bool}}{\Gamma;v\vdash e:\mathsf{supp}}\ (\text{Supp Bool}) \qquad \frac{\Gamma\vdash e:U \quad \Gamma\vdash\pi:\texttt{perm} \quad \Gamma\vdash e':T \quad T\ f\in\mathsf{fld}(U)}{\Gamma;v\vdash\texttt{PointsTo}(e\texttt{[}f\texttt{]},\pi,e'):\mathsf{supp}}\ (\text{Supp Points To})$$

$$\frac{\Gamma\vdash e:\texttt{Thread} \quad \Gamma\vdash\pi:\texttt{perm}}{\Gamma;v\vdash\texttt{Perm}(e\texttt{[join]},\pi):\mathsf{supp}}\ (\text{Supp Perm})$$

$$\frac{\Gamma\vdash e:T}{\Gamma;v\vdash\texttt{Pure}(e):\mathsf{supp}}\ (\text{Supp Perm}) \qquad \frac{\Gamma\vdash\pi:U \quad \Gamma\vdash\overline{\pi'}:\overline{T} \quad \mathsf{ptype}(\kappa^{\mathsf{grp}},U)=\mathit{fin}\ \mathsf{group}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}}{\Gamma;v\vdash\pi.\kappa^{\mathsf{grp}}\texttt{<}\overline{\pi'}\texttt{>}:\mathsf{supp}}\ (\text{Supp Pred})$$

$$\frac{\Gamma;v\vdash F:\mathsf{supp} \quad \Gamma;v\vdash F':\mathsf{supp}}{\Gamma;v\vdash F\ \mathit{lop}\ F':\mathsf{supp}}\ (\text{Supp Log Op}) \quad \mathit{lop}\in\{*,\&\}$$

(Supp Ex)
$$\frac{\Gamma \vdash T : \diamond \quad \Gamma, \alpha : T; v \vdash F : \mathsf{supp} \quad \Gamma, \alpha : T, \alpha' : T; v; \mathtt{true} \vdash' F \,\&\, F[\alpha'/\alpha] \mathbin{-\!\!*} \alpha == \alpha'}{\Gamma; v \vdash (\mathtt{ex}\ T\ \alpha)(F) : \mathsf{supp}}$$

We also define *weakly supported formulas* $(\Gamma; v \vdash_{\mathsf{w}} F : \mathsf{supp})$. Formulas that implement datagroups are required to be weakly supported.

$(\Gamma; v \vdash_{\mathsf{w}} F : \mathsf{supp})$ iff $\begin{cases} \text{this judgment is provable with the rules for } (\Gamma; v \vdash F : \mathsf{supp}), \text{ except} \\ \text{that in the last premise of (Supp Ex) the } \vdash' \text{ is replaced by } \vdash'_{\mathsf{w}} \end{cases}$

The formula that implements the `header` datagroup in the `List` class in Section A.3 is an example of a formula that is weakly supported but not supported.

Lemma 2 *If* $(\Gamma; v \vdash F : \mathsf{supp})$, *then* $(\Gamma; v \vdash \mathsf{split}(F) : \mathsf{supp})$.

Proof. Note that all rule and axiom schemes for logical consequence, except from the ones for applying class axioms and opening/closing abstract predicates, are "closed under splitting". This means that $(\Gamma; v; \bar{F} \vdash' G)$ implies $(\Gamma; v; \mathsf{split}(\bar{F}) \vdash' \mathsf{split}(G))$, by induction on the derivation. Given this observation, it is straightforward to prove Lemma 2 by induction on the structure of F. □

H Datagroup Formulas

For the soundness of the split/merge axiom for datagroups, it is important that splitting of datagroup formulas commutes with substitutions in the sense of of Lemma 3 below. (In that lemma, $\bar{\alpha}$ represent the parameters of a datagroup definition.) In order to guarantee this commutativity, we define a judgment that can be summarized as follows:

$\Gamma; \Gamma' \vdash F : \star$ $\begin{cases} \text{If } \alpha^{\mathsf{perm}} \text{ occurs freely in } F \text{ outside a type, then } \alpha^{\mathsf{perm}} \in \mathsf{dom}(\Gamma'). \\ \text{If } \alpha^{\mathsf{perm}} \text{ occurs freely in } F \text{ inside a type, then } \alpha^{\mathsf{perm}} \in \mathsf{dom}(\Gamma). \\ \text{All occurrences of the permission constant 1 in } F \text{ are inside types.} \end{cases}$

Split-parametric Specification Values, $\Gamma; \Gamma' \vdash \pi : T, \star$

($\star$ Val)
$$\frac{\Gamma, \Gamma' \vdash v : T}{\Gamma; \Gamma' \vdash v : T, \star}$$

($\star$ Val Var)
$$\frac{\Gamma, \Gamma' \vdash \alpha^{\mathsf{val}} : T}{\Gamma; \Gamma' \vdash \alpha^{\mathsf{val}} : T, \star}$$

($\star$ Perm Var)
$$\frac{\Gamma' \vdash \alpha^{\mathsf{perm}} : \mathtt{perm}}{\Gamma; \Gamma' \vdash \alpha^{\mathsf{perm}} : \mathtt{perm}, \star}$$

($\star$ Split)
$$\frac{\Gamma; \Gamma' \vdash \pi : \mathtt{perm}, \star}{\Gamma; \Gamma' \vdash \mathtt{split}(\pi) : \mathtt{perm}, \star}$$

Split-parametric Formulas, $\Gamma; \Gamma' \vdash F : \star$**:**

($\star$ Bool)
$$\frac{\Gamma, \Gamma' \vdash e : \mathtt{bool}}{\Gamma; \Gamma' \vdash e : \star}$$

($\star$ Points To)
$$\frac{T\ f \in \mathsf{fld}(U) \quad \Gamma, \Gamma' \vdash e : U \quad \Gamma; \Gamma' \vdash \pi : \mathtt{perm}, \star \quad \Gamma, \Gamma' \vdash e' : T}{\Gamma; \Gamma' \vdash \mathtt{PointsTo}(e[f], \pi, e') : \star}$$

($\star$ Perm)
$$\frac{\Gamma, \Gamma' \vdash e : \mathtt{Thread} \quad \Gamma; \Gamma' \vdash \pi : \mathtt{perm}, \star}{\Gamma; \Gamma' \vdash \mathtt{Perm}(e[\mathtt{join}], \pi) : \star}$$

($\star$ Pure)
$$\frac{\Gamma, \Gamma' \vdash e : T}{\Gamma; \Gamma' \vdash \mathtt{Pure}(e) : \star}$$

($\star$ Pred)
$$\frac{\Gamma,\Gamma' \vdash \pi : U \quad \Gamma;\Gamma' \vdash \overline{\pi}' : \overline{T},\star \quad \mathsf{ptype}(\kappa^{\mathsf{grp}},U) = \mathit{fin}\ \mathtt{group}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}}{\Gamma;\Gamma' \vdash \pi.\kappa^{\mathsf{grp}}\texttt{<}\overline{\pi}'\texttt{>} : \star}$$

($\star$ Log Op)
$$\frac{\Gamma;\Gamma' \vdash F : \star \quad \Gamma;\Gamma' \vdash F' : \star \quad \mathit{lop} \in \{*,\&\}}{\Gamma;\Gamma' \vdash F\ \mathit{lop}\ F' : \star}$$

($\star$ Ex)
$$\frac{\Gamma \vdash T : \diamond \quad \Gamma,\alpha : T;\Gamma' \vdash F : \star}{\Gamma;\Gamma' \vdash (\mathtt{ex}\ T\ \alpha)(F) : \star}$$

Lemma 3

(a) *If* $(\Gamma;\overline{\alpha} : \overline{T} \vdash \pi' : U,\star)$ *and* $(\Gamma,\overline{\alpha} : \overline{T} \vdash \overline{\pi} : \overline{T})$, *then* $\pi'[\mathsf{split}(\overline{\pi})/\overline{\alpha}] = \mathsf{split}(\pi'[\overline{\pi}/\overline{\alpha}])$.
(b) *If* $(\Gamma;\overline{\alpha} : \overline{T} \vdash F : \star)$ *and* $(\Gamma,\overline{\alpha} : \overline{T} \vdash \overline{\pi} : \overline{T})$, *then* $F[\mathsf{split}(\overline{\pi})/\overline{\alpha}] = \mathsf{split}(F[\overline{\pi}/\overline{\alpha}])$.

Proof. By inductions on $(\Gamma;\overline{\alpha} : \overline{T} \vdash \pi' : U,\star)$ and $(\Gamma;\overline{\alpha} : \overline{T} \vdash F : \star)$. □

We now define the judgment $(\Gamma;\Gamma';\nu \vdash F : \mathsf{grp})$ for datagroup formulas F as the conjunction of the two judgments that we have just defined:

$$\Gamma;\Gamma';\nu \vdash F : \mathsf{grp} \quad \text{iff} \quad (\Gamma,\Gamma';\nu \vdash_{\mathsf{w}} F : \mathsf{supp}) \text{ and } (\Gamma;\Gamma' \vdash F : \star)$$

Lemma 4 *If* $(\Gamma;\Gamma';\nu \vdash F : \mathsf{grp})$, *then* $(\Gamma;\Gamma';\nu \vdash \mathsf{split}(F) : \mathsf{grp})$.

Proof. By induction on the structure of F, using Lemma 3. The crucial observation is the following: if $(\Gamma,\overline{\alpha} : \overline{T},\alpha' : U,\alpha'' : U;\nu;\mathtt{true} \vdash'_{\mathsf{w}} F\ \&\ F[\alpha''/\alpha']\ \texttt{-*}\ \alpha' == \alpha'')$ and $\mathsf{split}(F) = F[\mathsf{split}(\overline{\alpha})/\overline{\alpha}]$, then it follows that $(\Gamma,\overline{\alpha} : \overline{T},\alpha' : U,\alpha'' : U;\nu;\mathtt{true} \vdash'_{\mathsf{w}} \mathsf{split}(F)\ \&\ \mathsf{split}(F)[\alpha''/\alpha']\ \texttt{-*}\ \alpha' == \alpha'')$, because logical consequence is closed under substitution (Lemma 49). □

I Method and Predicate Subtyping

Parkinson and Bierman [27] define a method subtyping relation (which they call *specification compatibility*) that is more liberal than standard behavioral subtyping. They make heavy use of this additional freedom in their examples. Their subtyping relation has the disadvantage that it is defined in terms of the Hoare triple judgment and involves a universal quantification over commands. We prefer to define method subtyping in terms of logical consequence, because we find that cleaner and also because a universal quantification over commands seems to be troublesome for algorithmic verification, which is our ultimate goal.

For convenience, we repeat the definition of method subtyping from Section 6.1:

Method Subtyping, $\Gamma \vdash MT <: MT'$ **and** $\Gamma \vdash \mathit{fin}\ MT <: \mathit{fin}'\ MT'$**:**

(Mth Sub) $\quad m \neq \mathtt{run}$
$$\frac{\overline{T}' <: \overline{T} \quad U <: U' \quad V_0 <: V_0' \quad \overline{V}' <: \overline{V} \quad G = (\mathtt{ex}\ U\ \alpha'')(G_0) \quad G' = (\mathtt{ex}\ U'\ \alpha'')(G_0') \quad \Gamma,\iota_0 : V_0;\iota_0;\mathtt{true} \vdash (\mathtt{fa}\ \overline{T}'\ \overline{\alpha})(\mathtt{fa}\ \overline{V}'\ \overline{\iota})(F' \texttt{-*} (\mathtt{ex}\ \overline{W}\ \overline{\alpha}')(F * (\mathtt{fa}\ U\ \alpha'')(G_0 \texttt{-*} G_0')))}{\Gamma \vdash \texttt{<}\overline{T}\ \overline{\alpha},\overline{W}\ \overline{\alpha}'\texttt{>}\mathtt{req}\ F;\mathtt{ens}\ G;U\ m(V_0\ \iota_0,\overline{V}\ \overline{\iota}) <: \texttt{<}\overline{T}'\ \overline{\alpha}\texttt{>}\mathtt{req}\ F';\mathtt{ens}\ G';U'\ m(V_0'\ \iota_0,\overline{V}'\ \overline{\iota})}$$

(Run Sub) $\quad G = (\mathtt{ex}\ \mathtt{void}\ \alpha)(G_0) \quad G' = (\mathtt{ex}\ \mathtt{void}\ \alpha)(G_0')$
$$\frac{V_0 <: V_0' \qquad \Gamma,\iota_0 : V_0;\iota_0;\mathtt{true} \vdash (F' \texttt{-*} F) * (\mathtt{fa}\ \mathtt{void}\ \alpha)(G_0 \texttt{-*} G_0')}{\Gamma \vdash \mathtt{req}\ F;\mathtt{ens}\ G;\mathtt{void}\ \mathtt{run}(V_0\ \iota_0) <: \mathtt{req}\ F';\mathtt{ens}\ G';\mathtt{void}\ \mathtt{run}(V_0'\ \iota_0)}$$

For qualified method types: $\quad \Gamma \vdash \mathit{fin}\ MT <: \mathit{fin}'\ MT'$ iff $\mathit{fin}' = \varepsilon \wedge \Gamma \vdash MT <: MT'$

Note that in (Mth Sub) there is a dependency of the postcondition of the supertype on the precondition of the supertype. For the `run`-method, this dependency would lead to unsoundness, because the pre- and post-conditions of `run` are used separately as precondition for `fork`, respectively, post-condition for `join`. This is why we have a more restrictive subtyping rule for `run`. Note also that the subtyping rule for `run` guarantees that `run`-methods have no logic parameters (because all `run`-method types are subtypes of `Thread.run`'s type, which has no logic parameters). Logic parameters for `run` would lead to unsoundness unless we enforced that they get instantiated uniformly at the `fork` and the `join` site.

We follow [27] and allow variable argument length for predicates. If a formula of the form $\pi.P\texttt{<}\overline{\pi}\texttt{>}$ misses arguments of types $\overline{T}'$, it is semantically equivalent to $\texttt{(ex }\overline{T}'\ \overline{\alpha}'\texttt{)(}\pi.P\texttt{<}\overline{\pi},\overline{\alpha}'\texttt{>)}$. As explained in [27], predicates with varargs are sometimes useful for flexible subclassing. For datagroups we prohibit varargs, because the existential quantification would render the merge axiom unsound.

Predicate Subtyping, $pt <: pt'$ and $fin\ pt <: fin'\ pt'$:

(Pred Sub)

$$\frac{}{\texttt{pred}\ P\texttt{<}\overline{T}\ \overline{\alpha},\overline{T}'\ \overline{\alpha}'\texttt{>} <: \texttt{pred}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}}$$

(Grp Sub)

$$\frac{}{\texttt{group}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} <: \texttt{group}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}}$$

For qualified predicate types: $fin\ pt <: fin'\ pt'$ iff $fin' = \varepsilon \wedge pt <: pt'$

J Class Axioms

We require that class axioms are proven with a restricted logical consequence judgment:

$$\vdash'' \ \triangleq\ \vdash'_{\mathsf{w}} \text{ without class axioms}$$

We disallow the application of class axioms for proving class axioms in order to avoid circularities. Recall that $\vdash'_{\mathsf{w}}$ is already a subsystem of $\vdash$, which restricts the merge axiom (see Section H). We disallow unrestricted merging so that it is sound to use class axioms in order to prove uniqueness of existential witnesses when showing that formulas are supported. This is sometimes needed, see the `List.header` datagroup in Section A.3.

$$C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{sound}$$
$$\text{iff}$$
$$\mathsf{axiom}(C\texttt{<}\overline{\alpha}\texttt{>}) = F \ \Rightarrow\ \overline{\alpha} : \overline{T}, \texttt{this} : C\texttt{<}\overline{\alpha}\texttt{>};\ \texttt{this};\ C\ \texttt{isclassof this} \vdash'' F$$

K Good Interfaces and Class Declarations

More auxiliary definitions:

$$\begin{array}{rcl}
fin\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \overline{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*}\texttt{\}} & \Rightarrow & \mathsf{methods}(C) \triangleq \mathsf{dom}(md\texttt{*}) \\
\texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ \overline{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*}\texttt{\}} & \Rightarrow & \mathsf{methods}(I) \triangleq \mathsf{dom}(mt\texttt{*}) \\
fin\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \overline{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*}\texttt{\}} & \Rightarrow & \mathsf{preds}(C) \triangleq \mathsf{dom}(pd\texttt{*}) \\
\texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ \overline{U}\ \texttt{\{}pt\texttt{*}\ ax\texttt{*}\ mt\texttt{*}\texttt{\}} & \Rightarrow & \mathsf{preds}(I) \triangleq \mathsf{dom}(pt\texttt{*}) \cup \{\texttt{state}, \texttt{init}\} \\
fin\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \overline{V}\ \texttt{\{}fd\texttt{*}\ pd\texttt{*}\ ax\texttt{*}\ md\texttt{*}\texttt{\}} & \Rightarrow & \mathsf{declared}(C) \triangleq \mathsf{dom}(fd\texttt{*})
\end{array}$$

In the following definitions, we conceive the partial functions mtype and ptype as total functions that map elements outside their domains to the special element undef. Furthermore, we extend the subtyping relation: $<: = \{(T,U) \mid T <: U\} \cup \{(\mathsf{undef},\mathsf{undef})\}$.

$$C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{extends}\ U \triangleq \begin{cases} U \text{ is a (parameterized) non-}\texttt{final}\text{ class} \\ f \in \mathsf{dom}(\mathsf{fld}(U)) \ \Rightarrow\ f \notin \mathsf{declared}(C) \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \overline{\alpha} : \overline{T} \vdash \mathsf{mtype}(m, C\texttt{<}\overline{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\overline{\alpha}\texttt{>}) <: pt) \end{cases}$$

$$I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ m \in \mathsf{methods}(I)) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ P \in \mathsf{preds}(I)) \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \overline{\alpha} : \overline{T} \vdash \mathsf{mtype}(m, I\texttt{<}\overline{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, I\texttt{<}\overline{\alpha}\texttt{>}) <: pt) \end{cases}$$

$$I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ \overline{U} \ \triangleq\ (\forall U \in \overline{U})(I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ U)$$

$$C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{implements}\ U \triangleq \begin{cases} U \text{ is a (parameterized) interface} \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \mathsf{mtype}(m, C\texttt{<}\overline{\alpha}\texttt{>}) \neq \mathsf{undef}) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\overline{\alpha}\texttt{>}) \neq \mathsf{undef}) \\ (\forall m, mt)(\mathsf{mtype}(m,U) = MT \ \Rightarrow\ \overline{\alpha} : \overline{T} \vdash \mathsf{mtype}(m, C\texttt{<}\overline{\alpha}\texttt{>}) <: MT) \\ (\forall P, pt)(\mathsf{ptype}(P,U) = pt \ \Rightarrow\ \mathsf{ptype}(P, C\texttt{<}\overline{\alpha}\texttt{>}) <: pt) \end{cases}$$

$$C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{implements}\ \overline{U} \ \triangleq\ (\forall U \in \overline{U})(C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{implements}\ U)$$

Good Predicate- and Method-Types, $\Gamma \vdash pt : \diamond$ and $\Gamma \vdash mt : \diamond$:

$$\text{(Pred Type)}\ \frac{\Gamma \vdash \overline{T} : \diamond}{\Gamma \vdash \mathit{pmod}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} : \diamond} \qquad \text{(Mth Type)}\ \frac{\begin{array}{c} m \in \{\texttt{run},\texttt{fork},\texttt{join}\} \Rightarrow \Gamma(\texttt{this}) <: \texttt{Thread} \\ \Gamma' = \Gamma, \overline{\alpha} : \overline{T}, \overline{\imath} : \overline{V} \quad \Gamma' \vdash \overline{T}, F, U, \overline{V} : \diamond \quad \Gamma'' = \Gamma', \texttt{result} : U \\ \Gamma'' \vdash G : \diamond \quad m = \texttt{run} \Rightarrow (\Gamma''; \texttt{this} \vdash G : \mathsf{supp} \wedge \texttt{this}[\texttt{join}] \notin G) \end{array}}{\Gamma \vdash \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\,\texttt{req}\ F; \texttt{ens}\ G;\ U\ m(\overline{V}\ \overline{\imath})\ : \diamond}$$

We remark that the supportedness of `run`'s postcondition is only needed to ensure soundness for multiple resource-splitting joiners. We could support both arbitrary postconditions and multiple joiners if we introduced a `run`-method modifier "`multi-join`" such that only `multi-join` `run`-methods may have multiple resource-splitting joiners and must have supported postconditions. It would also be important to require that methods that override `multi-join` methods are again `multi-join`.

Good Interfaces, $\mathit{int} : \diamond$:

$$\text{(Int)}\ \frac{I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{type\text{-}extends}\ \overline{U} \quad \texttt{init} \notin \mathsf{dom}(\mathit{pt}\texttt{*}) \qquad \overline{\alpha} : \overline{T} \vdash \overline{T}, \overline{U}, \mathit{pt}\texttt{*} : \diamond \quad \overline{\alpha} : \overline{T}, \texttt{this} : I\texttt{<}\overline{\alpha}\texttt{>} \vdash \mathit{ax}\texttt{*}, \mathit{mt}\texttt{*} : \diamond}{\texttt{interface}\ I\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\,\texttt{ext}\ \overline{U}\,\{\mathit{pt}\texttt{*}\ \mathit{ax}\texttt{*}\ \mathit{mt}\texttt{*}\}\ : \diamond} \qquad \text{(Ax)}\ \frac{\Gamma \vdash F : \diamond}{\Gamma \vdash \texttt{axiom}\ F : \diamond}$$

Our policy for interface extensions requires that every method or predicate declared in a superinterface is explicitly repeated in the subinterface, possibly with a more specific type. An exception is the special predicate `state`. If `state` is not explicitly declared in an interface, then `group state<perm`α`>` gets "automatically" included. (This is achieved by the rule (Ptype Interface Implicit)). Real Java has more policies

for avoiding repetition of method specifications in subinterfaces. These policies are called *inheritance of method signatures*. They are complicated by the fact that Java allows method overloading based on method signatures. We avoid this complexity, but do not consider this a substantial restriction because inherited method signatures can be filled in at compile time.

Good Classes, $cl : \diamond$**:**

$$\text{(Cls)}\ \frac{\overline{\alpha} : \overline{T} \vdash \overline{T}, U, \overline{V}, fd\text{*} : \diamond \quad \overline{\alpha} : \overline{T} \vdash pd\text{*} : \diamond \ \mathsf{in}\ C\texttt{<}\overline{\alpha}\texttt{>} \quad \overline{\alpha} : \overline{T}, \texttt{this} : C\texttt{<}\overline{\alpha}\texttt{>} \vdash ax\text{*}, md\text{*} : \diamond}{\mathit{fin}\ \texttt{class}\ C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{ext}\ U\ \texttt{impl}\ \overline{V}\ \{fd\text{*}\ pd\text{*}\ ax\text{*}\ md\text{*}\} : \diamond} \quad C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{implements}\ \overline{V} \quad C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \mathsf{sound} \quad \texttt{init} \notin \mathsf{dom}(pd\text{*})$$

$$\text{(Fld)}\ \frac{\Gamma \vdash T : \diamond}{\Gamma \vdash T\ f : \diamond} \qquad \text{(Grp Def)}\ \frac{\Gamma, \texttt{this} : U; \overline{\alpha} : \overline{T}; \texttt{this} \vdash F : \texttt{grp}}{\Gamma \vdash \mathit{fin}\ \texttt{group}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} = F : \diamond\ \mathsf{in}\ U} \qquad \text{(Grp Ext)}\ \frac{\Gamma, \texttt{this} : U; \overline{\alpha} : \overline{T}; \texttt{this} \vdash F : \texttt{grp}}{\Gamma \vdash \mathit{fin}\ \texttt{ext group}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{by}\ F : \diamond\ \mathsf{in}\ U}$$

$$\text{(Pred Def)}\ \frac{\Gamma, \texttt{this} : U, \overline{\alpha} : \overline{T} \vdash F : \diamond}{\Gamma \vdash \mathit{fin}\ \texttt{pred}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} = F : \diamond\ \mathsf{in}\ U} \qquad \text{(Pred Ext)}\ \frac{\Gamma, \texttt{this} : U, \overline{\alpha} : \overline{T} \vdash F : \diamond}{\Gamma \vdash \mathit{fin}\ \texttt{ext pred}\ P\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\ \texttt{by}\ F : \diamond\ \mathsf{in}\ U}$$

$$\text{(Mth)}\ \frac{\Gamma, \overline{\alpha} : \overline{T}, \overline{\imath} : \overline{V}; \texttt{this} \vdash \{F * \texttt{this} \neq \texttt{null}\} c : U\{(\texttt{ex}\ U\ \texttt{result})(G)\}}{\Gamma \vdash \mathit{fin}\ \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>req}\ F\texttt{; ens}\ G\texttt{;}\ U\ m(\overline{V}\ \overline{\imath})\{c\} : \diamond} \quad \Gamma \vdash \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>req}\ F\texttt{; ens}\ G\texttt{;}\ U\ m(\overline{V}\ \overline{\imath}) : \diamond$$

L Semantics of Expressions and Formulas

L.1 Semantics of Values

We define the set of *semantic values*:

$$\mu \in \mathsf{SemVal} \ \triangleq\ \mathsf{CIVal} \uplus (0,1]$$

The following typing rule extends the typing judgment for values to semantic values:

$$\frac{\mu \in (0,1]}{\Gamma \vdash \mu : \texttt{perm}}$$

Semantics of Specification Values, $[\![\pi]\!] \in \mathsf{SemVal}$**:**

$$[\![\texttt{null}]\!] \triangleq \texttt{null} \quad [\![o]\!] \triangleq o \quad [\![n]\!] \triangleq n \quad [\![b]\!] \triangleq b \quad [\![1]\!] \triangleq 1 \quad [\![\texttt{split}(\pi^{\mathsf{perm}})]\!] \triangleq \tfrac{1}{2}[\![\pi^{\mathsf{perm}}]\!]$$

We leave the semantics of read-only and logic variables undefined, because we deal with these variables by substitution.

Lemma 5 (Injectivity of Value Semantics) *If* $[\![\pi_1]\!] = [\![\pi_2]\!]$*, then* $\pi_1 = \pi_2$*.*

Proof. By induction on the structure of π_1. □

L.2 Semantics of Expressions

As explained in Section D, $[\![op]\!]$ is a function of type $\mathsf{Heap} \to \mathsf{SemVal}^{\mathsf{arity}(op)} \rightharpoonup \mathsf{SemVal}$ that is compatible with $\mathsf{types}(op)$. We require that $[\![op]\!]$ is invariant under heap extensions, field updates and applications of substitutions to dynamic types:

(a) If $[\![op]\!]^h(\overline{v}) = w$ and $h \subseteq h'$, then $[\![op]\!]^{h'}(\overline{v}) = w$.

(b) If $h' = h[o.f \mapsto u]$, then $[\![op]\!]^h = [\![op]\!]^{h'}$.

(c) If $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$ and $h' = h[\sigma]$, then $[\![op]\!]^h = [\![op]\!]^{h'}$.

Semantics of Expressions, $[\![e]\!] : \mathsf{Heap} \to \mathsf{Stack} \rightharpoonup \mathsf{SemVal}$**:**

$$\frac{[\![\pi]\!] = \mu}{[\![\pi]\!]^h_s = \mu}\ (\text{Sem Val}) \qquad \frac{s(\ell) = v}{[\![\ell]\!]^h_s = v}\ (\text{Sem Var}) \qquad \frac{[\![e]\!]^h_s = o \quad h(o)_2(f) = v}{[\![e.f]\!]^h_s = v}\ (\text{Sem Get})$$

$$\frac{[\![op]\!]^h(v_1,\ldots,v_n) = w \qquad [\![e_1]\!]^h_s = v_1 \quad \cdots \quad [\![e_n]\!]^h_s = v_n}{[\![op(e_1,\ldots,e_n)]\!]^h_s = w}\ (\text{Sem Op})$$

L.3 Semantic Validity of Boolean Expressions

Recall that our (mostly proof-theoretic) logical consequence judgment, via an axiom, depends on semantic validity of boolean expressions, $\Gamma \models e$. To define semantic validity formally, let σ range over *closing substitutions*, i.e, elements of $\mathsf{Var} \rightharpoonup \mathsf{ClVal}$. The following rule defines a judgment, $\Gamma \vdash \sigma : \diamond$, for *well-typed closing substitutions* σ:

$$\frac{\mathsf{dom}(\sigma) = \mathsf{dom}(\Gamma) \cap \mathsf{Var} \quad (\forall x \in \mathsf{dom}(\sigma))(\Gamma_{\mathsf{hp}} \vdash \sigma(x) : \Gamma(x)[\sigma])}{\Gamma \vdash \sigma : \diamond}$$

We say that a heap h is *total* iff for all o in $\mathsf{dom}(h)$ and all $f \in \mathsf{fld}(h(o)_1)$ it is the case that $f \in h(o)_2$. Now, we define $\Gamma \models e$ as follows:

$$\mathsf{ClosingSubst}(\Gamma) \triangleq \{\ \sigma \mid \Gamma \vdash \sigma : \diamond\ \} \qquad \mathsf{Heap}(\Gamma) \triangleq \{\ h \mid \Gamma_{\mathsf{hp}} \vdash h : \diamond \text{ and } h \text{ is total}\ \}$$

$$\Gamma \models e \quad \text{iff} \quad \begin{cases} \Gamma \vdash e : \mathtt{bool} \text{ and} \\ (\ \forall \Gamma' \supseteq_{\mathsf{hp}} \Gamma, h \in \mathsf{Heap}(\Gamma'), \sigma \in \mathsf{ClosingSubst}(\Gamma')\ )(\ [\![e[\sigma]]\!]^h_\emptyset = \mathtt{true}\ ) \end{cases}$$

L.4 Heap Joining

In this section, we state and prove some simple properties of heaps. For convenience, we repeat the definitions from Section 4.1.

We define a function that maps heaps to flat relations. To this end, let $\mathsf{HpDom} = \mathsf{ObjId} \times \mathsf{Ty} \times (\mathsf{FieldId} \times \mathsf{ClVal})_\perp$ with the following partial order: $(o,T,x) \leq (p,U,y)$ iff $(o,T,x) = (p,U,y)$ or $x = \perp$. For $X \subseteq \mathsf{HpDom}$, let $\downarrow X = \{y \mid (\exists x \in X)(y \leq x)\}$. A subset X of HpDom is called *downward closed* iff $\downarrow X = X$. A *functional relation* is a downward closed subset $\overline{h}$ of HpDom such that $(o,T,\perp),(o,T',\perp) \in \overline{h}$ implies $T = T'$ and $(o,T,(f,v)),(o,T,(f,v')) \in \overline{h}$ implies $v = v'$. Let FunRel be the set of all functional relations.

Lemma 6 (Complements Exist) *If* $\overline{h} \subseteq \overline{h}'$, *then* $\overline{h}' = \overline{h} \cup \downarrow(\overline{h}' \setminus \overline{h})$.

Proof. On the one hand, $\overline{h}' = \overline{h} \cup (\overline{h}' \setminus \overline{h}) \subseteq \overline{h} \cup \downarrow(\overline{h}' \setminus \overline{h})$. On the other hand, $(\overline{h}' \setminus \overline{h}) \subseteq \overline{h}'$, thus, $\downarrow(\overline{h}' \setminus \overline{h}) \subseteq \overline{h}'$ because $\overline{h}'$ is downward closed, thus $\overline{h} \cup \downarrow(\overline{h}' \setminus \overline{h}) \subseteq \overline{h}'$ because $\overline{h} \subseteq \overline{h}'$ by assumption. □

We define the following bijections:

$$\backslash\backslash : \mathsf{Heap} \to \mathsf{FunRel} \quad \backslash\backslash h \triangleq \{\ (o,T,x) \mid h(o)_1 = T \ \wedge\ x \in h(o)_2 \cup \{\bot\}\ \}$$

$$/\!/ : \mathsf{FunRel} \to \mathsf{Heap} \quad /\!/\overline{h}(o)_1 \triangleq T, \text{ if } (o,T,\bot) \in \overline{h}\ \ /\!/\overline{h}(o)_2 \triangleq \{(f,v) \mid (o,/\!/\overline{h}(o)_1,(f,v)) \in \overline{h}\}$$

Lemma 7 *$/\!/ \backslash\backslash h = h$ and $\backslash\backslash /\!/ \overline{h} = \overline{h}$.*

Proof. It suffices to show that $\backslash\backslash$ is injective and $\backslash\backslash /\!/ \overline{h} = \overline{h}$. Injectivity of $\backslash\backslash$ is easy to check. $\backslash\backslash /\!/ \overline{h} = \overline{h}$ holds for the following reason:

$$\begin{array}{ll} & (o,T,x) \in \backslash\backslash /\!/ \overline{h} \\ \text{iff} & /\!/\overline{h}(o)_1 = T \ \wedge\ x \in /\!/\overline{h}(o)_2 \cup \{\bot\} \\ \text{iff} & (o,T,\bot) \in \overline{h} \wedge (x \in \{\ (f,v) \mid (o,T,(f,v)) \in \overline{h}\ \} \ \vee\ x = \bot) \\ \text{iff} & ((o,T,\bot) \in \overline{h} \wedge x \in \{\ (f,v) \mid (o,T,(f,v)) \in \overline{h}\ \}) \ \vee\ ((o,T,x) \in \overline{h} \ \wedge\ x = \bot) \\ \text{iff} & x \in \{\ (f,v) \mid (o,T,(f,v)) \in \overline{h}\ \} \ \vee\ ((o,T,x) \in \overline{h} \ \wedge\ x = \bot) \\ \text{iff} & (o,T,x) \in \overline{h} \end{array}$$

□

We define a partial operator $*$ that joins heaps:

$$\# \triangleq \{\ (h,h') \mid \backslash\backslash h \cup \backslash\backslash h' \in \mathsf{FunRel}\ \} \qquad * : \# \to \mathsf{Heap} \qquad h*h' \triangleq /\!/(\backslash\backslash h \cup \backslash\backslash h')$$

We define a partial order on heaps:

$$h \leq h' \quad \text{iff} \quad \backslash\backslash h \subseteq \backslash\backslash h'$$

Lemma 8 *If $h \subseteq h'$, then $h \leq h'$.*

Lemma 9 *If $(\Gamma \vdash h : \diamond)$, $h' \leq h$ and $\mathsf{dom}(h') = \mathsf{dom}(h)$, then $(\Gamma \vdash h' : \diamond)$.*

Lemma 10 *If $(\Gamma \vdash h_1 : \diamond)$, $(\Gamma \vdash h_2 : \diamond)$, and $h_1 \# h_2$, then $(\Gamma \vdash h_1 * h_2 : \diamond)$.*

Lemma 11 *$*$ is commutative, associative and monotone with respect to $\leq$.*

Lemma 12 (Characterizeing $\leq$ in terms of $*$)

(a) *If $h \# h'$, then $h \leq h * h'$.*
(b) *If $h \leq h''$, then $h'' = h * h'$ for some h'.*

Proof. Part (a) holds because $\backslash\backslash h \subseteq \backslash\backslash h \cup \backslash\backslash h'$. For part (b), let $h' = /\!/ \downarrow (\backslash\backslash h'' \setminus \backslash\backslash h)$ and use Lemma 6. □

Sets of heaps have greatest lower bounds and if they have upper bounds at all, they have least upper bounds:

$$\textstyle\bigwedge_{i\in I} h_i \triangleq /\!/ \bigcap_{i\in I} \backslash\backslash h_i \qquad \text{If } \{h_i \mid i \in I\} \text{ has an upper bound: } \bigvee_{i\in I} h_i \triangleq /\!/ \bigcup_{i\in I} \backslash\backslash h_i$$

Lemma 13 (Infima and Bounded Suprema) (a) *$\bigwedge_{i\in I} h_i$ is the greatest lower bound of $\{h_i \mid i \in I\}$.*
(b) *If $\{h_i \mid i \in I\}$ has an upper bound, then $\bigvee_{i\in I} h_i$ is its least upper bound.*

Lemma 14 *If $[\![op]\!]^h(\overline{v}) = w$ and $h \leq h'$, then $[\![op]\!]^{h'}(\overline{v}) = w$.*

Proof. Our conditions on $[\![op]\!]$ were upwards closure with respect to $\subseteq$ and invariance under field updates. If $h \leq h'$, then there exists h'' such that $h'' \subseteq h'$ and h'' can be obtained from h by a sequence of field updates. □

We define the *domain extension operator* $h \nearrow h'$ as follows:

$$h \nearrow h' \triangleq /\!/(\backslash\backslash h \cup \{ (o, h'(o), \bot) \mid o \in \mathsf{dom}(h') \})$$

Lemma 15 $\mathsf{dom}(h \nearrow h') = \mathsf{dom}(h) \cup \mathsf{dom}(h')$

Lemma 16 *If* $h \leq h'$ *and* $(\Gamma' \vdash h' : \diamond)$, *then* $(\Gamma' \vdash h \nearrow h' : \diamond)$.

Lemma 17 (Domain Extension Properties) (a) $h \leq h \nearrow h'$
(b) *If* $h \leq h'$, *then* $h \nearrow h' \leq h'$.
(c) $h \nearrow h = h$
(d) *If* $h_1 \leq h_2$ *and* $h'_1 \leq h'_2$, *then* $h_1 \nearrow h'_1 \leq h_2 \nearrow h'_2$.
(e) $(h_1 * h_2) \nearrow h' = (h_1 \nearrow h') * (h_2 \nearrow h')$
(f) *If* $\mathsf{dom}(h') \subseteq \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$, *then* $h_1 * h_2 = (h_1 \nearrow h') * h_2$.

Lemma 18 (Field Update) *If* $h \# h'$, $o \in \mathsf{dom}(h), \mathsf{dom}(h')$ *and* $f \notin \mathsf{dom}(h'(o)_2)$, *then* $h[o.f \mapsto v] \# h'$ *and* $h[o.f \mapsto v] * h' = (h * h')[o.f \mapsto v]$.

L.5 Resource Joining

In this section, we state and prove some simple properties about resources. For convenience, we repeat the definitions from Section 4.1.

We define: A triple $(h, \mathscr{P}, \mathscr{Q}) \in \mathsf{Heap} \times \mathsf{PermTable} \times \mathsf{PermTable}$ is *sound* whenever the following conditions hold:

(a) $\mathsf{fst} \circ h \vdash h : \diamond$
(b) $\mathscr{P} \leq \mathscr{Q}$.
(c) For all $o \in \mathsf{dom}(h)$ and $f \in \mathsf{dom}(h(o)_2)$, either $\mathscr{P}(o,f) > 0$ or $\mathscr{Q}(o,f) < 1$.
(d) For all $o \notin \mathsf{dom}(h)$ and all f, $\mathscr{P}(o,f) = 0$ and $\mathscr{Q}(o,f) = 1$.
(e) For all o, f, if $\mathscr{Q}(o,f) < 1$ then $o \in \mathsf{dom}(h)$ and $f \in \mathsf{dom}(h(o)_2)$.

$$\mathscr{R} \in \mathsf{Resources} \triangleq \{ (h, \mathscr{P}, \mathscr{Q}) \mid (h, \mathscr{P}, \mathscr{Q}) \text{ is sound} \}$$

We define projections: For $\mathscr{R} = (h, \mathscr{P}, \mathscr{Q})$, let $\mathscr{R}_{\mathsf{hp}} = h$, $\mathscr{R}_{\mathsf{loc}} = \mathscr{P}$ and $\mathscr{R}_{\mathsf{glo}} = \mathscr{Q}$. We extend the compatibility relations as follows:

$$(h, \mathscr{P}, \mathscr{Q}) \# (h', \mathscr{P}', \mathscr{Q}') \quad \text{iff} \quad h \# h',\ \mathscr{P} \# \mathscr{P}',\ \mathscr{Q} = \mathscr{Q}',\ \text{and } (h * h', \mathscr{P} + \mathscr{P}', \mathscr{Q}) \text{ is sound}$$

Now, we define the resource joining operator $*$:

$$* : \# \to \mathsf{Resources} \qquad (h, \mathscr{P}, \mathscr{Q}) * (h', \mathscr{P}', \mathscr{Q}) \triangleq (h * h', \mathscr{P} + \mathscr{P}', \mathscr{Q})$$

We define an order on Resources as follows:

$$\mathscr{R} \leq \mathscr{R}' \quad \text{iff} \quad \mathscr{R}_{\mathsf{hp}} \leq \mathscr{R}'_{\mathsf{hp}},\ \mathscr{R}_{\mathsf{loc}} \leq \mathscr{R}'_{\mathsf{loc}} \text{ and } \mathscr{R}'_{\mathsf{glo}} = \mathscr{R}_{\mathsf{glo}}$$

For heap h and global permission table $\mathscr{Q}$, we define the subheap of h that consists of all its `final` fields:

$$h|\mathscr{Q} \triangleq /\!/\{ (o,T,x) \in \backslash\backslash h \mid x = \bot \text{ or } \mathscr{Q}(x) < 1 \} \qquad \mathsf{final}(h, \mathscr{P}, \mathscr{Q}) \triangleq (h|\mathscr{Q}, \mathbf{0}, \mathscr{Q})$$

Lemma 19 *If* $\mathscr{R} \in \mathsf{Resources}$, *then* $\mathsf{final}(\mathscr{R}) \in \mathsf{Resources}$ *and* $\mathsf{final}(\mathscr{R}) \leq \mathscr{R}$.

Lemma 20 $*$ *is commutative, associative and monotone.*

Proof. This follows from commutativity, associativity and monotonicity of heap joining and the permission table operation $+$. □

Lemma 21 (Characterization of $\leq$ in terms of $*$)

(a) *If $\mathscr{R}\#\mathscr{R}'$, then $\mathscr{R} \leq \mathscr{R} * \mathscr{R}'$.*
(b) *If $\mathscr{R} \leq \mathscr{R}''$, then $\mathscr{R}'' = \mathscr{R} * \mathscr{R}'$ for some $\mathscr{R}'$.*

Proof. Part (a) holds because $h \leq h * h'$ and $\mathscr{P} \leq \mathscr{P} + \mathscr{P}'$. For part (b), let $\mathscr{R} = (h, \mathscr{P}, \mathscr{Q}) \leq (h'', \mathscr{P}'', \mathscr{Q}) = \mathscr{R}''$. By Lemma 12, $h'' = h * h'$ for some h'. Define $\mathscr{R}' = (h', \mathscr{P}'' - \mathscr{P}, \mathscr{Q})$. □

Lemma 22 *If $(h, \mathscr{P}, \mathscr{Q})$ satisfies resource axioms (a), (b), (d) and (e) (but not necessarily (c)), then there exists a greatest resource $\mathscr{R}$ such that $\mathscr{R}_{\mathsf{hp}} \leq h$, $\mathscr{R}_{\mathsf{loc}} \leq \mathscr{P}$ and $\mathscr{R}_{\mathsf{glo}} = \mathscr{Q}$. We denote this as $(h, \mathscr{P}, \mathscr{Q})^\circ$.*

Proof. $\mathscr{R}_{\mathsf{loc}} = \mathscr{P}$, $\mathsf{dom}(\mathscr{R}_{\mathsf{hp}}) = \mathsf{dom}(h)$, $\mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} = \mathsf{fst} \circ h$, $\mathsf{snd} \circ \mathscr{R}_{\mathsf{hp}} = \mathsf{snd} \circ \mathscr{R}_{\mathsf{hp}} \setminus \{(o, f) \mid \mathscr{P}(o, f) = 0 \text{ and } \mathscr{Q}(o, f) = 1\}$. □

We define greatest lower and least upper bounds:

$$\begin{array}{l}\text{If } \mathsf{final}(\mathscr{R}^i) = \mathsf{final}(\mathscr{R}^j) \text{ for all } i, j \text{ in } I\text{:} \\ \quad \bigwedge_{i\in I} \mathscr{R}^i \triangleq (\bigwedge_{i\in I} \mathscr{R}^i_{\mathsf{hp}}, \bigwedge_{i\in I} \mathscr{R}^i_{\mathsf{loc}}, \bigwedge_{i\in I} \mathscr{R}^i_{\mathsf{loc}})^\circ \\ \text{If } \{\mathscr{R}^i \mid i \in I\} \text{ has an upper bound:} \\ \quad \bigvee_{i\in I} \mathscr{R}^i \triangleq (\bigvee_{i\in I} \mathscr{R}^i_{\mathsf{hp}}, \bigvee_{i\in I} \mathscr{R}^i_{\mathsf{loc}}, \bigvee_{i\in I} \mathscr{R}^i_{\mathsf{glo}})\end{array}$$

(Note that, if a set of resources has an upper bound or share the same final subresources, then the global permission tables must be equal. Thus, the operation on the global permission table in these definitions is trivial.)

Lemma 23 (Infima and Suprema) *Suppose $\{\mathscr{R}_i \mid i \in I\}$ has an upper bound. Then:*

(a) *If $\mathsf{final}(\mathscr{R}^i) = \mathsf{final}(\mathscr{R}^j)$ for all i, j in I, then $\bigwedge_{i\in I} \mathscr{R}_i$ is the greatest lower bound of $\{\mathscr{R}_i \mid i \in I\}$.*
(b) *If $\{\mathscr{R}_i \mid i \in I\}$ has an upper bound, then $\bigvee_{i\in I} \mathscr{R}_i$ is the least upper bound of $\{\mathscr{R}_i \mid i \in I\}$.*

We define resource splitting:

$$\tfrac{1}{2}(h, \mathscr{P}, \mathscr{Q}) \triangleq (h, \tfrac{1}{2}\mathscr{P}, \mathscr{Q})$$

Lemma 24 $\frac{1}{2}(\mathscr{R} * \mathscr{R}) = \mathscr{R}$.

We define the *domain extension operator* $\mathscr{R} \nearrow \mathscr{R}'$ as follows:

$$\mathscr{R} \nearrow \mathscr{R}' \triangleq (\mathscr{R}_{\mathsf{hp}} \nearrow \mathscr{R}'_{\mathsf{hp}}, \mathscr{R}_{\mathsf{loc}}, \mathscr{R}_{\mathsf{glo}})$$

Lemma 25 $\mathsf{dom}((\mathscr{R} \nearrow \mathscr{R}')_{\mathsf{hp}}) = \mathsf{dom}(\mathscr{R}_{\mathsf{hp}}) \cup \mathsf{dom}(\mathscr{R}'_{\mathsf{hp}})$

Lemma 26 *If $\mathscr{R} \leq \mathscr{R}'$ and $(\Gamma' \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond)$, then $(\Gamma' \vdash (\mathscr{R} \nearrow \mathscr{R}')_{\mathsf{hp}} : \diamond)$.*

Lemma 27 (Domain Extension Properties) (a) $\mathscr{R} \leq \mathscr{R} \nearrow \mathscr{R}'$
(b) *If $\mathscr{R} \leq \mathscr{R}'$, then $\mathscr{R} \nearrow \mathscr{R}' \leq \mathscr{R}'$.*
(c) $\mathscr{R} \nearrow \mathscr{R} = \mathscr{R}$
(d) *If $\mathscr{R}_1 \leq \mathscr{R}_2$ and $\mathscr{R}'_1 \leq \mathscr{R}'_2$, then $\mathscr{R}_1 \nearrow \mathscr{R}'_1 \leq \mathscr{R}_2 \nearrow \mathscr{R}'_2$.*

(e) $(\mathscr{R}_1 * \mathscr{R}_2) \nearrow \mathscr{R}' = (\mathscr{R}_1 \nearrow \mathscr{R}') * (\mathscr{R}_2 \nearrow \mathscr{R}')$.
(f) *If* $\mathsf{dom}(\mathscr{R}'_{\mathsf{hp}}) \subseteq \mathsf{dom}((\mathscr{R}_1)_{\mathsf{hp}}) \cup \mathsf{dom}((\mathscr{R}_2)_{\mathsf{hp}})$, *then* $\mathscr{R}_1 * \mathscr{R}_2 = (\mathscr{R}_1 \nearrow \mathscr{R}') * \mathscr{R}_2$.

Proof. These are obvious consequences of the definition of $\nearrow$ and the fact that the domain extension operator for heaps satisfies the same properties (by Lemma 17). □

$$\mathscr{R}[o.f \mapsto v] \triangleq (\mathscr{R}_{\mathsf{hp}}[o.f \mapsto v], \mathscr{R}_{\mathsf{loc}}, \mathscr{R}_{\mathsf{glo}})$$

Lemma 28 (Field Update) *Suppose* $\mathscr{R}_{\mathsf{loc}}(o,f) = 1$, $T\ f \in \mathsf{fld}(\mathscr{R}_{\mathsf{hp}}(o)_1)$ *and* $\mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash v : T$. *Then:*

(a) $\mathscr{R}[o.f \mapsto v] \in \mathsf{Resources}$
(b) *If* $\mathscr{R} \# \mathscr{R}'$ *and* $o \in \mathsf{dom}(\mathscr{R}'_{\mathsf{hp}})$,
then $\mathscr{R}[o.f \mapsto v] \# \mathscr{R}'$ *and* $\mathscr{R}[o.f \mapsto v] * \mathscr{R}' = (\mathscr{R} * \mathscr{R}')[o.f \mapsto v]$.

Proof. For part (a), it is easy to check that the resource axioms hold for $\mathscr{R}[o.f \mapsto v]$. For part (b), note that $\mathscr{R}'_{\mathsf{loc}}(o,f) = 0$ (because $\mathscr{R}_{\mathsf{loc}} \# \mathscr{R}'_{\mathsf{loc}}$ and $\mathscr{R}_{\mathsf{loc}}(o,f) = 1$) and $\mathscr{R}'_{\mathsf{glo}}(o,f) = \mathscr{R}_{\mathsf{glo}}(o,f) \geq \mathscr{R}_{\mathsf{loc}}(o,f) = 1$. Therefore, $f \notin \mathsf{dom}(h'(o)_2)$ by resource axiom (c). Then $\mathscr{R}_{\mathsf{hp}}[o.f \mapsto v] \# \mathscr{R}'$ and $\mathscr{R}_{\mathsf{hp}}[o.f \mapsto v] * \mathscr{R}'_{\mathsf{hp}} = (\mathscr{R}_{\mathsf{hp}} * \mathscr{R}'_{\mathsf{hp}})[o.f \mapsto v]$, by Lemma 18. It is easy to check that the resource axioms hold for $((\mathscr{R}_{\mathsf{hp}} * \mathscr{R}'_{\mathsf{hp}})[o.f \mapsto v],\ \mathscr{R}_{\mathsf{loc}} + \mathscr{R}'_{\mathsf{loc}},\ \mathscr{R}_{\mathsf{glo}})$. □

L.6 Predicate Environments

In this section, we repeat the definition of predicate environments from Section 4.2 in a slightly more general form. We call these more general functions *predicate pre-environments*. Predicate pre-environments have type $\prod \kappa \in X.\mathsf{Dom}(\kappa) \to \mathbb{2}$, where $X \subseteq \mathsf{Pred}(ct)$. *Predicate environments* are predicate pre-environments where $X = \mathsf{Pred}(ct)$. *Pre*-environments are auxiliary entities for constructing least fixed points of endofunctions on predicate environments (see Section L.10). At the top level, we are only interested in predicate environments (where $X = \mathsf{Pred}(ct)$).

Definition 3 (Predicate Pre-Environments) Let $X \subseteq \mathsf{Pred}(ct)$. A *predicate pre-environment over* X is a function of type $\prod \kappa \in X.\mathsf{Dom}(\kappa) \to \mathbb{2}$ such that the following axioms hold:

(a) If $(\overline{\pi}, \mathscr{R}, r, \overline{\pi}'), (\overline{\pi}, \mathscr{R}', r, \overline{\pi}') \in \mathsf{Dom}(\kappa)$ and $\mathscr{R} \leq \mathscr{R}'$,
then $\mathscr{E}(\kappa)(\overline{\pi}, \mathscr{R}, r, \overline{\pi}') \leq \mathscr{E}(\kappa)(\overline{\pi}, \mathscr{R}', r, \overline{\pi}')$.
(b) If $(\overline{\pi}, \mathscr{R}, r, \overline{\pi}') \in \mathsf{Dom}(\kappa)$ and $\mathsf{final}(\mathscr{R}_i) = \mathsf{final}(\mathscr{R}_j)$ for all i, j in I,
then $\mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)(\overline{\pi}, \bigwedge_{i \in I} \mathscr{R}_i, r, \overline{\pi}') = \bigwedge_{i \in I} \mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)(\overline{\pi}, \mathscr{R}_i, r, \overline{\pi}')$.
(c) If $(\overline{\pi}, \mathscr{R}, r, \overline{\pi}') \in \mathsf{Dom}(P^{\mathsf{grp}}\mathtt{@}C)$,
then $\mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)(\overline{\pi}, \mathscr{R}, r, \overline{\pi}') \leq \mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)(\overline{\pi}, \frac{1}{2}\mathscr{R}, r, \mathsf{split}(\overline{\pi}'))$.
(d) If $(\overline{\pi}, \mathscr{R}_1, r, \overline{\pi}'), (\overline{\pi}, \mathscr{R}_2, r, \overline{\pi}') \in \mathsf{Dom}(P^{\mathsf{grp}}\mathtt{@}C)$,
then $\mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)(\overline{\pi}, \mathscr{R}_1, r, \mathsf{split}(\overline{\pi}')) \wedge \mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)(\overline{\pi}, \mathscr{R}_2, r, \mathsf{split}(\overline{\pi}')) \leq \mathscr{E}(P^{\mathsf{grp}}\mathtt{@}C)$
$(\overline{\pi}, \mathscr{R}_1 * \mathscr{R}_2, r, \overline{\pi}')$.
(e) If $(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi}'), (\overline{\pi}, (h', \mathscr{P}, \mathscr{Q}'), r, \overline{\pi}') \in \mathsf{Dom}(\kappa)$, $o \in \mathsf{dom}(h)$, $\mathscr{P}(o,f) = 0$, $\mathscr{Q}(o,f) = 1$, $\mathscr{Q}' = \mathscr{Q}[(o,f) \mapsto 0]$ and $h' = h[o.f \mapsto v]$,
then $\mathscr{E}(\kappa)(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi}') \leq \mathscr{E}(\kappa)(\overline{\pi}, (h', \mathscr{P}, \mathscr{Q}'), r, \overline{\pi}')$.
(f) If $(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi}'), (\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}'), r, \overline{\pi}') \in \mathsf{Dom}(\kappa)$, $o \in \mathsf{dom}(h)$, $\mathscr{P}(o, \mathtt{join}) \leq x \leq \mathscr{Q}(o, \mathtt{join})$, and $\mathscr{Q}' = \mathscr{Q}[(o, \mathtt{join}) \mapsto x]$,
then $\mathscr{E}(\kappa)(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}), r, \overline{\pi}') \leq \mathscr{E}(\kappa)(\overline{\pi}, (h, \mathscr{P}, \mathscr{Q}'), r, \overline{\pi}')$.

A *predicate environment* is a predicate pre-environment over $\mathsf{Pred}(ct)$.

Lemma 29 *The set of all predicate pre-environments over X (with the order inherited from the underlying function space) is a complete lattice.*

Proof. It suffices to show that the (pointwise) greatest lower bound of a set of predicate pre-environments over X is again a predicate pre-environment over X. Axiom (a) holds because pointwise infima of monotone functions are again monotone. Axiom (b) holds because pointwise infima of infima-preserving functions are again infima-preserving. The other axioms hold because $\bigwedge_{i\in I} f_i(x) \leq \bigwedge_{i\in I} f_i(y)$ if $f_i(x) \leq f_i(y)$ for all i in I. □

L.7 Semantics of Formulas

We define an auxiliary relation $(\Gamma \vdash \mathscr{R}, s : \diamond)$, where $\mathscr{R}$ is a resource and s a thread-local stack. Intuitively, $(\Gamma \vdash \mathscr{R}, s : \diamond)$ holds whenever $\mathscr{R}$ and s are type-compatible and furthermore $\mathsf{dom}(\Gamma)$ does not contain read-only or logic variables (as these are handled by substitution). Formally, let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ whenever the following statements hold:

- $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$
- $\Gamma_{\mathsf{hp}} \vdash \mathscr{R}_{\mathsf{hp}} : \diamond$
- $\Gamma \vdash s : \diamond$

Let $(\Gamma \vdash \mathscr{R}, s, F : \diamond)$ whenever the following statements hold:

- $\Gamma \vdash \mathscr{R}, s : \diamond$
- $\Gamma \vdash F : \diamond$

The relation $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$ is the unique subset of $(\Gamma \vdash \mathscr{R}, s, F : \diamond)$ that satisfies the clauses from Section 4.3.

L.8 Semantic Entailment

For reference, we repeat the definitions of semantic entailment from Section 5. First, we defined a semantic counterpart to the syntactic purity judgment:

$$\mathscr{Q}; h; s \models \bar{F} : \checkmark \quad \text{iff} \quad \mathscr{Q}; h; s \models \mathsf{pure}(e) \text{ for all field selection subexpressions } e \text{ of } \bar{F}$$

Then we defined *semantic entailment*. (As usual, we let $F_1 * \cdots * F_0 = \mathtt{true}$.)

$$\begin{array}{rll} \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F : \checkmark & \text{iff} & (\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F \text{ and } \mathscr{R}_{\mathsf{glo}}; \mathscr{R}_{\mathsf{hp}}; s \models F : \checkmark) \\ \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F_1, \ldots, F_n : \checkmark & \text{iff} & \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F_1 * \cdots * F_n : \checkmark \\ \Gamma \vdash \mathscr{E}; \bar{F} \models G : \checkmark & \text{iff} & (\forall \Gamma, \mathscr{R}, s)(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} : \checkmark \;\Rightarrow\; \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G : \checkmark) \end{array}$$

The following variations of semantic entailment are also useful:

$$\begin{array}{rll} \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F_1, \ldots, F_n & \text{iff} & \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F_1 * \cdots * F_n \\ \Gamma \vdash \mathscr{E}; \bar{F} \models G & \text{iff} & (\forall \Gamma, \mathscr{R}, s)(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} \;\Rightarrow\; \Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G) \\ \Gamma \vdash \mathscr{E}; \bar{F} \models_{\leq \mathscr{R}} G & \text{iff} & (\forall \Gamma, \mathscr{R}' \leq \mathscr{R}, s)(\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models \bar{F} \;\Rightarrow\; \Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models G) \end{array}$$

L.9 Interlude: A Relaxed Fixed Point Theorem

Predicate environments in the previous sections were abstract and not related to predicate definitions. In this and the next section, we show how to construct predicate environments that satisfy the predicate definitions from the class table. This is not entirely straightforward because predicate definitions can be circular. In order to deal with circularities, Parkinson and Bierman [27] forbid that predicate definitions contain predicates in negative positions. As a result of this restriction, Parkinson/Bierman's predicate definitions give rise to monotone functionals on predicate environments, and Parkinson/Bierman appeal to Tarski's fixed point theorem for the existence of a solution. Unfortunately, entirely disallowing predicates in negative positions is too restrictive for us, because the definition of the `ready`-predicate in our `Iterator`-example mentions the `state`-predicate of a `Node` in a negative position:

```
final class ListIterator<perm p, Collection iteratee>
   implements Iterator<p,iteratee>
{  ...
   pred ready = (ex Node x) ( ... (x.state<p> -* iteratee.state<p>) );
   ... }
```

Fortunately, the `state`-predicate does not depend on the `ready`-predicate, so that the negative occurrence in this example is not part of a cycle. In general, we can guarantee well-foundedness of predicate definitions if no cyclic dependency contains a negative predicate occurrence. We will make this precise in Section L.10.

First, we present some general fixed point theory. We denote the least element of a complete lattice L by $\bot_L$ (often omitting the subscript L). For complete lattices L and L', let $L \to L'$ be the set of all (not necessarily monotone) functions from L to L', and $L \xrightarrow{\mathsf{m}} L'$ be the set of all monotone functions from L to L'. We will make use of the following fixed point theorem.

Theorem 6 (Fixed Point Theorem) *If L is a complete lattice and $F \in L \xrightarrow{\mathsf{m}} L$, then f has a least fixed point.*

In our applications of this theorem, the complete lattice is a function space $X \to L$ where X is some set. We want to relax the fixed point theorem so that we can deal with certain non-monotone functionals in $(X \to L) \to (X \to L)$. Some definitions:

- For $F \in (X \to L) \to (X \to L)$ and $Y \subseteq X$, we say Y *is F-closed* whenever $f_{|Y} = g_{|Y}$ implies $F(f)_{|Y} = F(g)_{|Y}$ for all f, g in $X \to L$.
- For $F \in (X \to L) \to (X \to L)$ and $Y \subseteq X$, we define the restriction $\mathsf{r}(F,Y)$ of F to $Y \to L$ as follows:

$$\begin{array}{rcl} \mathsf{r}(F,Y) & : & (Y \to L) \to (Y \to L) \\ \mathsf{r}(F,Y)(f)(y) & \triangleq & F(f \cup \{\,(x,\bot) \mid x \in X \setminus Y\,\})(y) \end{array}$$

Lemma 30 *If $F \in (X \to L) \to (X \to L)$, Y is an F-closed subset of X and $f \in X \to L$, then $F(f)_{|Y} = \mathsf{r}(F,Y)(f_{|Y})$.*

Proof. Let $F \in (X \to L) \to (X \to L)$, and Y be an F-closed subset of X, and $f \in X \to L$. Let $f' = f_{|Y} \cup \{\,(x,\bot) \mid x \in X \setminus Y\,\}$. Clearly, $f_{|Y} = f'_{|Y}$.

$$\begin{array}{rcll} F(f)_{|Y} & = & F(f')_{|Y} & \text{(because } Y \text{ is } F\text{-closed)} \\ & = & \mathsf{r}(F,Y)(f_{|Y}) & \text{(by definition of } \mathsf{r}) \end{array}$$

□

Lemma 31 *If $F \in (X \to L) \to (X \to L)$ and $Z \subseteq Y \subseteq X$ and Z if F-closed, then Z is $\mathsf{r}(F,Y)$-closed.*

Proof. Suppose $F \in (X \to L) \to (X \to L)$ and $Z \subseteq Y \subseteq X$ and Z if F-closed. Let $f,g \in (Y \to L) \to (Y \to L)$ such that $f_{|Z} = g_{|Z}$. Define $f' = f \cup \{\,(x,\bot) \mid x \in X \setminus Y\,\}$ and $g' = g \cup \{\,(x,\bot) \mid x \in X \setminus Y\,\}$. Clearly, $f'_{|Z} = g'_{|Z}$.

$$\begin{array}{rcll} \mathsf{r}(F,Y)(f)_{|Z} & = & F(f')_{|Y|Z} & \text{(by definition of } \mathsf{r}(F,Y)) \\ & = & F(f')_{|Z} & \\ & = & F(g')_{|Z} & \text{(because } Z \text{ is } F\text{-closed)} \\ & = & F(g')_{|Y|Z} = \mathsf{r}(F,Y)(g)_{|Z} & \end{array}$$

□

Lemma 32 *If $F \in (X \to L) \to (X \to L)$ and $Z \subseteq Y \subseteq X$ and Z,Y are F-closed, then $\mathsf{r}(F,Z) = \mathsf{r}(\mathsf{r}(F,Y),Z)$.*

Proof. Let $F \in (X \to L) \to (X \to L)$ and $Z \subseteq Y \subseteq X$ and Z,Y be F-closed. Let $f \in Z \to L$ and $z \in Z$. Let $f' = f \cup \{\,(y,\bot) \mid y \in Y \setminus Z\,\}$ and $f'' = f \cup \{\,(x,\bot) \mid x \in X \setminus Z\,\}$

$$\begin{array}{rcll} \mathsf{r}(\mathsf{r}(F,Y),Z)(f)(z) & = & \mathsf{r}(F,Y)(f')(z) & \text{(by definition of } \mathsf{r}) \\ & = & \mathsf{r}(F,Y)(f''_{|Y})(z) & \\ & = & F(f'')(z) & \text{(by Lemma 30)} \\ & = & \mathsf{r}(F,Z)(f)(z) & \text{(by definition of } \mathsf{r}) \end{array}$$

□

For $Y \subseteq X$, the function $\mathsf{Lift}_{Y,X}$ is defined as follows:

$$\begin{array}{rcl} \mathsf{Lift}_{Y,X} & : & ((X \to L) \to X \to L) \to (Y \to L) \to (X \to L) \to X \to L \\ \mathsf{Lift}_{Y,X}(F)(g)(f)(x) & \triangleq & \begin{cases} g(x) & \text{if } x \in Y \\ F(f)(x) & \text{if } x \notin Y \end{cases} \end{array}$$

Definition 4 (Monotone Chains) Given $F \in (X \to L) \to (X \to L)$. An *$F$-monotone chain* is an ascending chain $\emptyset = X_0 \subseteq \cdots \subseteq X_n = X$ of F-closed sets such that for every i in $\{1,\ldots,n\}$ and g in $X_{i-1} \to L$ the function $\mathsf{Lift}_{X_{i-1},X_i}(\mathsf{r}(F,X_i))(g)$ is monotone.

Theorem 7 (Relaxed Fixed Point Theorem) *If L is a complete lattice, $F \in (X \to L) \to (X \to L)$ and an F-monotone chain exists, then F has a fixed point.*

Proof. Suppose L is a complete lattice and $F \in (X \to L) \to (X \to L)$. Let $\emptyset = X_0 \subseteq \cdots \subseteq X_n = X$ be an F-monotone chain. We show the following statement by induction on i:

For all i in $\{0,\ldots,n\}$, $\mathsf{r}(F,X_i)$ has a least fixed point.

For $i = 0$, this is trivial, because $\emptyset \to L$ is a singleton set. Let $i > 0$. By induction hypothesis, $\mathsf{r}(F,X_{i-1})$ has a least fixed point, call it g. Because $\mathsf{Lift}_{X_{i-1},X_i}(\mathsf{r}(F,X_i))(g)$ is monotone, this function, too, has a fixed point by the fixed point theorem 6. Call this fixed point f. We will now show that f is a fixed point of $\mathsf{r}(F,X_i)$.

First, let $x \in X_i \setminus X_{i-1}$. Then:

$$\begin{array}{lcll} \mathsf{r}(F,X_i)(f)(x) & = & \mathsf{Lift}_{X_{i-1},X_i}(\mathsf{r}(F,X_i))(g)(f)(x) & \text{(by definition of } \mathsf{Lift}_{X_{i-1},X_i}) \\ & = & f(x) & \text{(because } f \text{ is fixed point)} \end{array}$$

Using the fact that f is a fixed point and the definition of $\mathsf{Lift}_{X_{i-1},X_i}$, we can also show:

$$g = f_{|X_{i-1}}$$

Let $g' = g \cup \{\,(x,\bot) \mid x \in X_i \setminus X_{i-1}\,\}$. Then $f_{|X_{i-1}} = g'_{|X_{i-1}}$. Because X_{i-1} is $\mathsf{r}(F,X_i)$-closed (by Lemma 31), we obtain:

$$\mathsf{r}(F,X_i)(f)_{|X_{i-1}} = \mathsf{r}(F,X_i)(g')_{|X_{i-1}}$$

We also have:

$$\begin{array}{lcll} g & = & \mathsf{r}(F,X_{i-1})(g) & \text{(because } g \text{ is fixed point)} \\ & = & \mathsf{r}(\mathsf{r}(F,X_i),X_{i-1})(g) & \text{(by Lemma 32)} \\ & = & \mathsf{r}(\mathsf{r}(F,X_i),X_{i-1})(g'_{|X_{i-1}}) & \\ & = & \mathsf{r}(F,X_i)(g')_{|X_{i-1}} & \text{(by Lemma 30)} \end{array}$$

So, we have $\mathsf{r}(F,X_i)(f)_{|X_{i-1}} = \mathsf{r}(F,X_i)(g')_{|X_{i-1}} = g = f_{|X_{i-1}}$. □

L.10 Predicate Definitions

We define a functional $\mathscr{F}_{ct}$ that maps predicate environments to predicate environments:

$$\frac{\begin{array}{l} \mathsf{pbody}(r.P\texttt{<}\overline{\pi'}\texttt{>},C\texttt{<}\overline{\pi}\texttt{>}) = F \texttt{ ext } D\texttt{<}\overline{\pi''}\texttt{>} \\ C \neq \texttt{Object} \text{ and } \mathsf{arity}(P,D) = n \;\Rightarrow\; F' = r.P@D\texttt{<}\overline{\pi}'_{\mathsf{to}\,n}\texttt{>} \\ C = \texttt{Object} \text{ or } P \text{ is rooted in } C \;\Rightarrow\; F' = \texttt{true} \end{array}}{\mathscr{F}_{ct}(\mathscr{E})(P@C)(\overline{\pi},\mathscr{R},r,\overline{\pi'}) = \begin{cases} 1 & \text{if } \mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \mathscr{E};\mathscr{R};\emptyset \models F * F' \\ 0 & \text{otherwise} \end{cases}}$$

Lemma 33 (Well-Typedness of $\mathscr{F}_{ct}$) *If $\mathscr{E}$ is a pre-environment over X, then so is $\mathscr{F}_{ct}(\mathscr{E})$.*

Proof. We need to show that $\mathscr{F}_{ct}(\mathscr{E})$ satisfies the axioms for predicate environments. They are consequences of lemmas that we will prove later: Axiom (a) is a consequence of Lemma 71. Axiom (b) is a consequence of Lemma 83. Axiom (c) is a consequence of Lemma 75 and Lemma 3. Axiom (d) is a consequence of Lemma 4, Lemma 82 and Lemma 3. Axiom (e) is a consequence of Lemma 77. □

We want to impose a condition on the predicate definitions in *ct* that guarantees that $\mathscr{F}_{ct}$ has a fixed point. We formulate this condition in terms of the dependency graph that records dependencies between predicates. Basically, a predicate $P@C$ depends on $Q@D$ if P's definition in C mentions $Q@D$. We have to be a bit careful, though, to soundly account for subclassing and predicate inheritance.

We label dependencies $P@C \xrightarrow{\sigma} Q@D$ by a sign $\sigma \in \{+,-\}$. A negative label indicates that $Q@D$'s occurrence in $P@C$'s definition is in a negative position, i.e., as the left descendant of an odd number of implication nodes. To define the dependency graph formally, we first define a relation $F \xrightarrow{\sigma} P@C$, where σ ranges over $\{+,-\}$. Intuitively, $F \xrightarrow{\sigma} P@C$ holds whenever F's validity depends on $P@C$. The relation is defined by induction on the structure of F:

$$
\begin{array}{rcll}
\pi.P@C<\bar{\pi}'> & \xrightarrow{+} & P@C & \text{iff} \quad P@C \in \mathsf{Pred}(ct) \\
\pi.P<\bar{\pi}'> & \xrightarrow{+} & P@C & \text{iff} \quad P@C \in \mathsf{Pred}(ct) \\
F \ lop \ G & \xrightarrow{\sigma} & \kappa & \text{iff} \quad F \xrightarrow{\sigma} \kappa \text{ or } G \xrightarrow{\sigma} \kappa \quad \text{for } lop \in \{*, \&, |\} \\
F \mathrel{-\!\!*} G & \xrightarrow{\sigma} & \kappa & \text{iff} \quad F \xrightarrow{-\sigma} \kappa \text{ or } G \xrightarrow{\sigma} \kappa \\
(qt\ T\ \alpha)\,(F) & \xrightarrow{\sigma} & \kappa & \text{iff} \quad F \xrightarrow{\sigma} \kappa
\end{array}
$$

The second clause of the definition conservatively accounts for the possibility that subclasses can extend predicate definitions. If, for instance, π has class C then we have to record that $\pi.P<\bar{\pi}>$ does not only depend on $P@C$ but also on $P@C'$ for all subclasses C' of C. Our definition even says that $\pi.P<\bar{\pi}>$ depends on $P@C'$ *for all classes* C' *where* P *is defined*. This is a conservative approximation. As a result, our dependency graphs sometimes records spurious dependencies that do not really exist.

The following two rules define a relation $P@C \xrightarrow{\sigma} Q@D \subseteq \mathsf{Pred}(ct) \times \{+,-\} \times \mathsf{Pred}(ct)$. We view this relation as a σ-labeled directed graph on $\mathsf{Pred}(ct)$ and refer to it as *ct*'s *dependency graph*, $\mathsf{DepGraph}(ct)$.

The Dependency Graph, $P@C \xrightarrow{\sigma} Q@D$**:**

$$
\frac{P@D \in \mathsf{Pred}(ct) \quad C \preceq D}{P@C \xrightarrow{+} P@D}\ (\text{Dep Pred Sup})
\qquad
\frac{\mathsf{pbody}(P<\bar{\alpha}>, C<\bar{\pi}>) = F \quad F \xrightarrow{\sigma} \kappa}{P@C \xrightarrow{\sigma} \kappa}\ (\text{Dep Pred Def})
$$

The rule (Dep Pred Sup) accounts for the fact that predicates P defined in C implicitly depend on $P@D$, if D is a superclass of C that defines P. This is so because P's definition in C gets $*$-conjoined with P's definition in D.

An edge in $\mathsf{DepGraph}(ct)$ is *positive* (resp. *negative*) whenever its label is $+$ (resp. $-$). A path is *positive* whenever all its edges are positive.

> **Requirement:** *In legal class tables ct, all cycles in* $\mathsf{DepGraph}(ct)$ *must be positive.*

Theorem 8 (Existence of Predicate Environments) *If ct is legal, then there exists a predicate environment* $\mathscr{E}$ *such that* $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$.

In the remainder of this section, we will prove this theorem. First some auxiliary definitions: We write $\kappa \rightarrow^* \kappa'$ iff there is a path from κ to κ' in $\mathsf{DepGraph}(ct)$, and $\kappa \xrightarrow{-}{}^* \kappa'$ iff there is a path with at least one negative edge. Note that the graph $\xrightarrow{-}{}^*$ is acyclic, if *ct* is legal. For $X \subseteq \mathsf{Pred}(ct)$, we define $\downarrow X = \{\kappa \mid (\exists \kappa' \in X)(\kappa' \rightarrow^* \kappa)\}$. Furthermore, $\downarrow F = \{\kappa \mid (\exists \sigma, \kappa')(F \xrightarrow{\sigma} \kappa' \rightarrow^* \kappa)\}$.

To prove Theorem 8, we want to apply the relaxed fixed point theorem (Theorem 7). We formulate a concrete criterion for $\mathscr{F}_{ct}$-closedness.

Lemma 34 *If* $\downarrow F \subseteq X$ *and* $\mathscr{E}_{|X} = \mathscr{E}'_{|X}$, *then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$ *iff* $(\Gamma \vdash \mathscr{E}'; \mathscr{R}; s \models F)$.

Proof. By induction on the structure of F. □

Lemma 35 (Criterion for $\mathscr{F}_{ct}$**-Closedness)** *If* $\downarrow X \subseteq X$, *then* X *is* $\mathscr{F}_{ct}$*-closed.*

Proof. Let $\downarrow X \subseteq X$, $\mathscr{E}_{|X} = \mathscr{E}'_{|X}$ and $\kappa \in \mathsf{Pred}(ct)$. By inspecting the definition of $\mathscr{F}_{ct}$ and the definition of the dependency graph, we can see that there exists F such that $\downarrow F \subseteq \downarrow X$ and the following statements hold:

$$\mathscr{F}_{ct}(\mathscr{E})(\kappa)(\overline{\pi})(\mathscr{R},r)(\overline{\pi}') = 1 \quad \text{iff} \quad \mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \mathscr{E};\mathscr{R};s \models F$$
$$\mathscr{F}_{ct}(\mathscr{E}')(\kappa)(\overline{\pi})(\mathscr{R},r)(\overline{\pi}') = 1 \quad \text{iff} \quad \mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \mathscr{E}';\mathscr{R};s \models F$$

Because $\downarrow X \subseteq X$, we know that $\downarrow F \subseteq \downarrow X \subseteq X$ and, thus, $\mathscr{E}_{|\downarrow F} = \mathscr{E}'_{|\downarrow F}$. Now, by Lemma 34, it follows that $\mathscr{F}_{ct}(\mathscr{E})(\kappa)(\overline{\pi})(\mathscr{R},r)(\overline{\pi}') = \mathscr{F}_{ct}(\mathscr{E}')(\kappa)(\overline{\pi})(\mathscr{R},r)(\overline{\pi}')$. □

We say that *F is positive (resp. negative) on X* whenever $F \xrightarrow{\sigma} \kappa \in X$ implies $\sigma = +$ (resp. $\sigma = -$).

Lemma 36 (Positive Formulas are Monotone, Negative Formulas are Antitone) *Let $\downarrow F \subseteq X_0 \uplus X$, $\mathsf{dom}(\mathscr{E}_0) = X_0$, $\mathsf{dom}(\mathscr{E}) = \mathsf{dom}(\mathscr{E}') = X$ and $\mathscr{E} \leq \mathscr{E}'$.*

(a) *If F is positive on X and $(\Gamma \vdash \mathscr{E}_0 \cup \mathscr{E};\mathscr{R};s \models F)$, then $(\Gamma \vdash \mathscr{E}_0 \cup \mathscr{E}';\mathscr{R};s \models F)$.*
(b) *If F is negative on X and $(\Gamma \vdash \mathscr{E}_0 \cup \mathscr{E}';\mathscr{R};s \models F)$, then $(\Gamma \vdash \mathscr{E}_0 \cup \mathscr{E};\mathscr{R};s \models F)$.*

Proof. Simultaneously, by induction on the structure of F. □

For $X \subseteq \mathsf{Pred}(ct)$, we say that is *$X$ is positive* whenever $X \ni \kappa \xrightarrow{\sigma} \kappa' \in X$ implies $\sigma = +$.

Lemma 37 (Criterion for Monotonicity of Lift) *If $Y \subseteq X$ and $X \setminus Y$ is positive, then $\mathsf{Lift}_{Y,X}(\mathsf{r}(\mathscr{F}_{ct},X))(\mathscr{E}_0)$ is monotone for all pre-environments $\mathscr{E}_0$ over Y.*

Proof. Suppose $Y \subseteq X$, and $X \setminus Y$ is positive, and $\mathscr{E}_0$ is a pre-environment over Y. Extend $\mathscr{E}_0$ to a pre-environment $\mathscr{E}'_o$ over $Y \cup (\mathsf{Pred}(ct) \setminus X)$ as follows: $\mathscr{E}'_0(\kappa) = \mathscr{E}_0(\kappa)$ if $\kappa \in Y$, and $\mathscr{E}'_0(\kappa) = \bot$ if $\kappa \in \mathsf{Pred}(ct) \setminus X$. Note now that the following holds for all $\mathscr{E}$ over X and κ in $Y \cup (\mathsf{Pred}(ct) \setminus X)$:

$$\mathsf{Lift}_{Y,X}(\mathsf{r}(\mathscr{F}_{ct},X))(\mathscr{E}_0)(\mathscr{E})(\kappa) = \mathscr{E}'_0(\kappa)$$

Now consider $\mathscr{E}$ over X and κ in $X \setminus Y$. By inspecting the definition of $\mathscr{F}_{ct}$ and the definition of the dependency graph, we can see that there exists F such that F is positive on $X \setminus Y$ and the following holds:

$$\mathsf{Lift}_{Y,X}(\mathsf{r}(\mathscr{F}_{ct},X))(\mathscr{E}_0)(\mathscr{E})(\kappa)(\overline{\pi})(\mathscr{R},r)(\overline{\pi}') = 1 \quad \text{iff} \quad \mathsf{fst} \circ \mathscr{R}_{\mathsf{hp}} \vdash \mathscr{E}'_o \cup \mathscr{E};\mathscr{R};s \models F$$

Therefore, the monotonicity of $\mathsf{Lift}_{Y,X}(\mathsf{r}(\mathscr{F}_{ct},X)(\mathscr{E}_0)$ follows from Lemma 36. □

Proof of Theorem 8. *If ct is legal, then there exists a predicate environment $\mathscr{E}$ such that $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$.*

Proof. Suppose *ct* is legal, i.e., all cycles in $\mathsf{DepGraph}(ct)$ are positive. By the relaxed fixed point theorem (Theorem 7) it suffices to construct an $\mathscr{F}_{ct}$-monotone chain $\emptyset = X_0 \subseteq \cdots \subseteq X_n = \mathsf{Pred}(ct)$. The chain we construct has the property that $\downarrow X_i = X_i$ for all chain members X_i, which implies that they are $\mathscr{F}_{ct}$-closed (by Lemma 35). Clearly, $\downarrow \emptyset = \emptyset$. Suppose, we have constructed a chain $\emptyset = X_0 \subseteq \cdots \subseteq X_i \neq \mathsf{Pred}(ct)$. We want to construct the $(i+1)$-st member. Because $\xrightarrow{-}{}^*$ is acyclic, we know that $\mathsf{Pred}(ct) \setminus X_i$ has

a non-empty subset of vertices κ such that κ has no outgoing $\bar{\rightarrow}^*$-edge whose target is in $\mathsf{Pred}(ct) \setminus X_i$. Let S be the set of all such κ's, and let $X_{i+1} = X_i \cup \downarrow S$. Clearly, $\downarrow X_{i+1} \subseteq X_{i+1}$. It remains to be shown that $\mathsf{Lift}_{X_i,X_{i+1}}(\mathsf{r}(\mathscr{F}_{ct}, X_{i+1}))(\mathscr{E}_i)$ is monotone for all pre-environments $\mathscr{E}_i$ over X_i. By Lemma 37, it suffices to show that $(\downarrow S) \setminus X_i$ is positive. So let $\kappa \xrightarrow{\sigma} \kappa'$ and $\kappa, \kappa' \in (\downarrow S) \setminus X_i$. Because $\kappa \in \downarrow S$, there is a $\kappa'' \in S$ such that $\kappa'' \rightarrow^* \kappa$. If it was the case that $\sigma = -$, then $\kappa'' \bar{\rightarrow}^* \kappa'$, contradicting the minimality of κ''. Therefore, $\sigma = +$. □

M Basic Properties of Typing Judgments

Lemma 38 (Good Environments) *Let $\mathscr{J}$ range over right-hand sides of the forms $T : \diamond$, $v : T$, $\pi : T$, $e : T$, $F : \diamond$, $s : \diamond$, $obj : \diamond$ and $h : \diamond$. If $(\Gamma \vdash \mathscr{J})$, then $(\Gamma \vdash \diamond)$.*

Proof. By induction on the derivation of $(\Gamma \vdash \mathscr{J})$. □

Lemma 39 (Weakening) *Let $\mathscr{J}$ range over right-hand sides of the forms $T : \diamond$, $v : T$, $\pi : T$, $e : T$, $F : \diamond$, $s : \diamond$ and $obj : \diamond$. If $(\Gamma \vdash \mathscr{J})$, $\Gamma \subseteq \Gamma'$ and $(\Gamma' \vdash \diamond)$, then $(\Gamma' \vdash \mathscr{J})$.*

Proof. By induction on the derivation of $(\Gamma \vdash \mathscr{J})$. □

Note that the heap typing judgment $(\Gamma \vdash h : \diamond)$ does not satisfy weakening. This is intentional. We want that every object identifier o in Γ's domain represents some actual object identifier (= memory address) at runtime, whose dynamic type is $\Gamma(o)$.

Lemma 40 (Strengthening) *Let $\mathscr{J}$ range over right-hand sides of the forms $\diamond$, $U : \diamond$, $v : U$, $\pi : U$, $e : U$, $F : \diamond$ and $obj : \diamond$. If $(\Gamma, x : T \vdash \mathscr{J})$ and $x \notin \mathsf{fv}(\Gamma, \mathscr{J})$, then $(\Gamma \vdash \mathscr{J})$.*

Proof. By induction on the derivation of $(\Gamma, x : T \vdash \mathscr{J})$. □

Lemma 41 (Substitutivity and Inverse Substitutivity for Subtyping)

(a) *If $T <: U$, then $T[\sigma] <: U[\sigma]$.*
(b) *If $T[\sigma] <: U$, then $U = U'[\sigma]$ for some U'.*
(c) *If $T[\sigma] <: U[\sigma]$, then $T <: U$.*

Proof. All three parts by induction on the derivation of the subtyping judgment. The proof of part (c) uses part (b) to deal with the transitivity rule. □

Lemma 42 (Substitutivity) *Let $\mathscr{J}$ range over right-hand-sides of the forms $T : \diamond$, $v : U$, $\pi : U$, $e : U$ and $F : \diamond$.*

(a) *If $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{x}])$ and $(\Gamma, \overline{x} : \overline{T} \vdash \mathscr{J})$, then $(\Gamma[\overline{\pi}/\overline{x}] \vdash \mathscr{J}[\overline{\pi}/\overline{x}])$.*
(b) *If $(\Gamma \vdash \overline{e} : \overline{T})$ and $(\Gamma, \overline{\ell} : \overline{T} \vdash \mathscr{J})$, then $(\Gamma \vdash \mathscr{J}[\overline{e}/\overline{\ell}])$.*
(c) *If $(\Gamma \vdash \sigma : \diamond)$ and $(\Gamma \vdash \mathscr{J})$, then $(\Gamma_{\mathsf{hp}} \vdash \mathscr{J}[\sigma])$.*[5]

Proof. Part (a) by induction on $(\Gamma, \overline{x} : \overline{T} \vdash \mathscr{J})$. Part (b) by induction on $(\Gamma, \overline{\ell} : \overline{T} \vdash \mathscr{J})$. Part (c) follows from part (a) in the following way: Suppose $(\Gamma \vdash \sigma : \diamond)$ and $(\Gamma \vdash \mathscr{J})$. Let $\overline{x} = \mathsf{dom}(\Gamma) \cap \mathsf{Var}$. If $\overline{x} = \emptyset$, then $\Gamma_{\mathsf{hp}} = \Gamma$, $\sigma = \emptyset$ and $\mathscr{J}[\sigma] = \mathscr{J}$. $(\Gamma_{\mathsf{hp}} \vdash \mathscr{J}[\sigma])$ trivially follows. So suppose $\overline{x} \neq \emptyset$. Then $(\Gamma_{\mathsf{hp}} \vdash \sigma(\overline{x}) : \Gamma(\overline{x})[\sigma])$, by definition of $(\Gamma \vdash \sigma : \diamond)$. In particular, it follows that $(\Gamma_{\mathsf{hp}} \vdash \diamond)$ (by Lemma 38) and therefore $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$. Therefore, $\Gamma_{\mathsf{hp}}[\sigma] = \Gamma_{\mathsf{hp}}$ and, thus, $(\Gamma_{\mathsf{hp}}[\sigma] \vdash \sigma(\overline{x}) : \Gamma(\overline{x})[\sigma])$. Now, we can apply part (a) to obtain $(\Gamma_{\mathsf{hp}} = \Gamma_{\mathsf{hp}}[\sigma] \vdash \mathscr{J}[\sigma])$. □

[5] Recall that σ ranges over closed substitutions. See Section L.3 for the definition of $\Gamma \vdash \sigma : \diamond$.

Lemma 43 (Inverse Substitutivity for Values) *If* $(\Gamma \vdash \sigma : \Gamma')$ *and* $(\Gamma[\sigma] \vdash v : T[\sigma])$, *then* $(\Gamma, \Gamma' \vdash v : T)$.

Proof. In case v is an integer, boolean or `null`, this is obvious. So suppose that v is an object identifier or a read-only variable. Then $v \in \mathsf{dom}(\Gamma)$ and $\Gamma[\sigma](v) <: T[\sigma]$. But then $\Gamma(v) <: T$, by Lemma 41. But then $(\Gamma, \Gamma' \vdash v : T)$. □

Lemma 44 *If* $(\Gamma \vdash T : \diamond)$ *and* $T <: U$, *then* $(\Gamma \vdash U : \diamond)$.

Proof. By induction on $<:$, using substitutivity (Lemma 42) to deal with the type parameters of reference types. □

N Basic Properties of Logical Consequence

Lemma 45 (Well-Typedness) *If* $(\Gamma; v; \overline{F} \vdash G)$, *then* $(\Gamma; v \vdash \overline{F}, G : \diamond)$.

Proof. By inductions on the derivations. □

Lemma 46 (Weakening Validity of Boolean Expressions) *If* $\Gamma \models e$, $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$, *then* $\Gamma' \models e$.

Proof. This holds because, by definition, $\Gamma \models e$ entails that e is true in all heaps that extend Γ_{hp}, and, moreover, the truth of e does not depend on variables outside $\mathsf{fv}(e)$. □

Lemma 47 (Weakening) (a) *If* $(\Gamma; v; \overline{F} \vdash G)$, $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$, *then* $(\Gamma'; v; \overline{F} \vdash G)$.
(b) *If* $(\overline{F} \vdash e : \checkmark)$, *then* $(\overline{F}, \overline{E} \vdash e : \checkmark)$.
(c) *If* $(\overline{F} \vdash G : \checkmark)$, *then* $(\overline{F}, \overline{E} \vdash G : \checkmark)$.
(d) *If* $(\Gamma; v; \overline{F} \vdash G)$ *and* $(\Gamma \vdash \overline{E} : \diamond)$, *then* $(\Gamma; v; \overline{F}, \overline{E} \vdash G)$.

Proof. Parts (b) and (c) are immediate from the definitions. Parts (a) and (d) by inductions on the derivation of $(\Gamma; v; \overline{F} \vdash G)$. □

Lemma 48 (Substitutivity for Validity of Boolean Expressions) *If* $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{x}])$ *and* $(\Gamma, \overline{x} : \overline{T} \models e)$, *then* $(\Gamma[\overline{\pi}/\overline{x}] \models e[\overline{\pi}/\overline{x}])$.

Proof. Let $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : T[\overline{\pi}/\overline{x}])$ and $(\Gamma, \overline{x} : \overline{T} \models e)$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma[\overline{\pi}/\overline{\alpha}]$, $(\Gamma'_{\mathsf{hp}} \vdash h : \diamond)$ and $(\Gamma' \vdash \sigma : \diamond)$. We need to show that $[\![e[\overline{\pi}/\overline{x}][\sigma]]\!]^h_\emptyset = \mathtt{true}$. To this end, let $\sigma' = \sigma[\overline{x} \mapsto \overline{\pi}[\sigma]]$. Note that $e[\overline{\pi}/\overline{x}][\sigma] = e[\sigma']$. Therefore, we are done if we can show that $[\![e[\sigma']]\!]^h_\emptyset = \mathtt{true}$. Let $\overline{y} = \mathsf{dom}(\Gamma \setminus \Gamma_{\mathsf{hp}})$. Let $\Gamma'' = (\Gamma'_{\mathsf{hp}}, \overline{y} : \Gamma(\overline{y}), \overline{x} : \overline{T})$. Because $(\Gamma, \overline{x} : \overline{T} \models e)$, we know that $(\Gamma, \overline{x} : \overline{T} \vdash e : \mathtt{bool})$, thus, $(\Gamma_{\mathsf{hp}} \vdash \diamond)$, thus, $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$, thus, $\Gamma[\overline{\pi}/\overline{x}]_{\mathsf{hp}} = \Gamma_{\mathsf{hp}}$, thus, $\Gamma'' \supseteq_{\mathsf{hp}} (\Gamma, \overline{x} : \overline{T})$. Moreover, $(\Gamma''_{\mathsf{hp}} \vdash h : \diamond)$, because $\Gamma''_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$ and $(\Gamma'_{\mathsf{hp}} \vdash h : \diamond)$. Because $(\Gamma, \overline{x} : \overline{T} \models e)$, it therefore suffices to show that $(\Gamma'' \vdash \sigma' : \diamond)$:

Let first $y \in \overline{y}$. Because $(\Gamma' \vdash \sigma : \diamond)$, we know that $(\Gamma'_{\mathsf{hp}} \vdash \sigma(y) : \Gamma'(y)[\sigma])$. We also know that $\Gamma'_{\mathsf{hp}} = \Gamma''_{\mathsf{hp}}$ and $\sigma(y) = \sigma'(y)$ and $\Gamma'(y)[\sigma] = \Gamma[\overline{\pi}/\overline{x}](y)[\sigma] = \Gamma(y)[\overline{\pi}/\overline{x}][\sigma] = \Gamma(y)[\sigma']$. Hence, $\Gamma''_{\mathsf{hp}} \vdash \sigma'(y) : \Gamma(y)[\sigma']$.

Let now $x \in \overline{x}$ such that $\sigma'(x) = \pi[\sigma]$ and $\Gamma''(x) = T$. We know that $(\Gamma' \vdash \sigma : \diamond)$ and $(\Gamma' \vdash \pi : T[\overline{\pi}/\overline{x}])$. By substitutivity (Lemma 42(c)), it follows that $(\Gamma'_{\mathsf{hp}} \vdash \pi[\sigma] : T[\overline{\pi}/\overline{x}][\sigma])$. We know that $\Gamma'_{\mathsf{hp}} = \Gamma''_{\mathsf{hp}}$ and $\pi[\sigma] = \sigma'(x)$ and $T[\overline{\pi}/\overline{x}][\sigma] = T[\sigma']$. Therefore, $\Gamma''_{\mathsf{hp}} \vdash \sigma'(x) : T[\sigma']$. □

Lemma 49 (Substitutivity) (a) *If* $(\overline{F} \vdash e : \checkmark)$, *then* $(\overline{F}[\overline{\pi}/\overline{x}] \vdash e[\overline{\pi}/\overline{x}] : \checkmark)$.

(b) *If* $(\overline{F} \vdash G : \checkmark)$, *then* $(\overline{F}[\overline{\pi}/\overline{x}] \vdash G[\overline{\pi}/\overline{x}] : \checkmark)$.
(c) *If* $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{x}])$ *and* $(\Gamma, \overline{x} : \overline{T}; v; \overline{F} \vdash G)$, *then* $(\Gamma; v; \overline{F})[\overline{\pi}/\overline{x}] \vdash G[\overline{\pi}/\overline{x}]$.

Proof. (a) by induction on the structure of e, (b) by induction on the structure of G, and (c) by induction on the derivation of $(\Gamma, \overline{x} : \overline{T}; v; \overline{F} \vdash G)$. □

Lemma 50 (Specialization of Variable Types) *If* $(\Gamma, x : U; v; \overline{F} \vdash G)$, $(\Gamma, x : T \vdash \diamond)$ *and* $T <: U$, *then* $(\Gamma, x : T; v; \overline{F} \vdash G)$.

Proof. This follows from substitutivity (Lemma 49) and $(\Gamma, x : T \vdash x : U)$. □

Lemma 51 (Purity) *If* $(\Gamma; v; \overline{F} \vdash G)$, *then* $(\overline{F} \vdash G : \checkmark)$.

Proof. By induction on the derivation of $(\Gamma; v; \overline{F} \vdash G)$. For the proof cases (Ex Intro) and (Fa Elim), we need the syntactic restriction on quantified formulas $(qt\ T\ \alpha)(F)$, namely, that field selection expressions $e.f$ that occur in F must not contain occurrences of α. □

Lemma 52 (Cut) (a) *If* $(\Gamma; v; \overline{E} \vdash F)$ *and* $(F, \overline{G} \vdash e : \checkmark)$, *then* $(\overline{E}, \overline{G} \vdash e : \checkmark)$.
(b) *If* $(\Gamma; v; \overline{E} \vdash F)$ *and* $(F, \overline{G} \vdash \overline{H} : \checkmark)$, *then* $(\overline{E}, \overline{G} \vdash \overline{H} : \checkmark)$.
(c) *If* $(\Gamma; v; \overline{E} \vdash F)$ *and* $(\Gamma; v; F, \overline{G} \vdash H)$, *then* $(\Gamma; v; \overline{E}, \overline{G} \vdash H)$.

Proof. (a) and (b) are consequences of Lemma 51. Part (c) by induction on the derivation of $(\Gamma; v; F, \overline{G} \vdash H)$. □

Lemma 53 (`ispartof` is a Partial Order) *Suppose* $\Gamma; v \vdash F : \diamond$.

(a) $(\Gamma; v; \texttt{true} \vdash F\ \texttt{ispartof}\ F)$.
(b) $(\Gamma; v; \texttt{true} \vdash F\ \texttt{ispartof}\ H)$ *is derivable from* $(\Gamma; v; \texttt{true} \vdash F\ \texttt{ispartof}\ G)$ *and* $(\Gamma; v; \texttt{true} \vdash G\ \texttt{ispartof}\ H)$.

The derivations only use the rules for $*$, $-\!\!*$ *and the identity rule.*

Proof. Straightforward natural deduction proofs. □

O Basic Properties of Method Subtyping

We define a judgment, $\Gamma \vdash MT : \diamond$, for *well-formed method types*:

$$\Gamma \vdash \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>req}\ F; \texttt{ens}\ G; U\ m(\overline{V}\ \overline{\iota}) : \diamond \quad \text{iff} \quad \Gamma, \overline{\alpha} : \overline{T}, \overline{\iota} : \overline{V} \vdash \overline{T}, F, G, U, \overline{V} : \diamond$$

Lemma 54 (Method Subtyping is a Preorder) (a) *If* $(\Gamma \vdash MT : \diamond)$, *then* $(\Gamma \vdash MT <: MT)$.
(b) *If* $(\Gamma \vdash MT <: MT')$ *and* $(\Gamma \vdash MT' <: MT'')$, *then* $(\Gamma \vdash MT <: MT'')$.

Proof. By application of the natural deduction rules. For transitivity, in order to deal with the variant types of the self-parameter ι_0, one uses the fact that logical consequence is contravariant in the types of the free variables (Lemma 50). □

Lemma 55 (Substitutivity) *If* $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{x}])$ *and* $(\Gamma, \overline{x} : \overline{T} \vdash MT <: MT')$, *then* $\Gamma[\overline{\pi}/\overline{x}] \vdash MT[\overline{\pi}/\overline{x}] <: MT'[\overline{\pi}/\overline{x}]$.

Proof. This is a consequence of substitutivity for value subtyping and logical consequence (Lemmas 41 and 49). □

Lemma 56 $\mathsf{mtype}(m, T)[\overline{\pi}/\overline{\alpha}] = \mathsf{mtype}(m, T[\overline{\pi}/\overline{\alpha}])$

$$t\langle\overline{T}\,\overline{\alpha}\rangle <:_1 s\langle\overline{\pi}\rangle \triangleq t\langle\overline{T}\,\overline{\alpha}\rangle \texttt{ ext } s\langle\overline{\pi}\rangle \in ct \text{ or } t\langle\overline{T}\,\overline{\alpha}\rangle \texttt{ impl } s\langle\overline{\pi}\rangle \in ct$$

Lemma 57 *If* $t\langle\overline{T}\,\overline{\alpha}\rangle <:_1 s\langle\overline{\pi}\rangle$, $(\Gamma \vdash \overline{\pi'} : \overline{T}[\overline{\pi'}/\overline{\alpha}])$ *and* $\mathsf{mtype}(m, s\langle\overline{\pi}\rangle)$ *is defined, then* $(\Gamma \vdash \mathsf{mtype}(m, t\langle\overline{\pi'}\rangle) <: \mathsf{mtype}(m, s\langle\overline{\pi}[\overline{\pi'}/\overline{\alpha}]\rangle))$.

Proof. Because $ct : \diamond$, we know that $\overline{\alpha} : \overline{T} \vdash \mathsf{mtype}(m, t\langle\overline{\alpha}\rangle) <: \mathsf{mtype}(m, s\langle\overline{\pi}\rangle)$. Then $\Gamma \vdash \mathsf{mtype}(m, t\langle\overline{\pi'}\rangle) <: \mathsf{mtype}(m, s\langle\overline{\pi}[\overline{\pi'}/\overline{\alpha}]\rangle)$, by Lemmas 55 and 56. □

Lemma 58 (Monotonicity of mtype**)** *If* $T <: U$, $(\Gamma \vdash T : \diamond)$ *and* $\mathsf{mtype}(m, U)$ *is defined, then* $\mathsf{mtype}(m, T) <: \mathsf{mtype}(m, U)$.

Proof. By induction on the derivation of $T <: U$, where we use a "tight" transitivity rule: $T <:_1 U$, $U <: V \Rightarrow T <: V$. This transitivity rule gives rise to the same subtyping relation. The proof makes use of Lemma 57 □

P Basic Properties of Hoare Triples

Lemma 59 (Well-Typedness) (a) *If* $(\Gamma; v \vdash \{F\} hc \{G\})$, *then* $(\Gamma; v \vdash F, G : \diamond)$.
(b) *If* $(\Gamma; v \vdash \{F\} c : T \{G\})$, *then* $(\Gamma; v \vdash F, T, G : \diamond)$.

Proof. For *hc* by inspection of the last rule. For *c* by induction on the structure of *c*. □

Lemma 60 (Weakening) (a) *If* $(\Gamma; v \vdash \{F\} hc \{G\})$, $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$, *then* $(\Gamma'; v \vdash \{F\} hc \{G\})$.
(b) *If* $(\Gamma; v \vdash \{F\} c : T \{G\})$, $\Gamma \subseteq \Gamma'$ *and* $(\Gamma' \vdash \diamond)$, *then* $(\Gamma'; v \vdash \{F\} c : T \{G\})$.

Proof. For *hc* by inspection of the last rule. For *c* by induction on the structure of *c*. □

Lemma 61 (Substitutivity)

(a) *If* $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{x}])$ *and* $(\Gamma, \overline{x} : \overline{T}; v \vdash \{F\} hc \{G\})$, *then* $((\Gamma; v)[\overline{\pi}/\overline{x}] \vdash \{F[\overline{\pi}/\overline{x}]\} hc[\overline{\pi}/\overline{x}] \{G[\overline{\pi}/\overline{x}]\})$.
(b) *If* $(\Gamma[\overline{\pi}/\overline{x}] \vdash \overline{\pi} : T[\overline{\pi}/\overline{x}])$ *and* $(\Gamma, \overline{x} : T; v \vdash \{F\} c : U \{G\})$, *then* $((\Gamma; v)[\overline{\pi}/\overline{x}] \vdash \{F[\overline{\pi}/\overline{x}]\} c[\overline{\pi}/\overline{x}] : U[\overline{\pi}/\overline{x}] \{G[\overline{\pi}/\overline{x}]\})$.

Proof. For *hc* by inspection of the last rule. For *c* by induction on the structure of *c*. □

Lemma 62 (Logical Consequence) *If* $(\Gamma; v; F \vdash F')$ *and* $(\Gamma; v \vdash \{F'\} c : T \{G\})$, *then* $(\Gamma; v \vdash \{F\} c : T \{G\})$.

Proof. By induction on the structure of *c*. □

Lemma 63 (Subsumption) *If* $(\Gamma; v \vdash \{F\} c : T \{G\})$ *and* $T <: U$, *then* $(\Gamma; v \vdash \{F\} c : U \{G\})$.

Proof. By induction on the structure of *c*. □

We abbreviate $(\exists H)(\Gamma; v \vdash \{F\} hc \{H\} \wedge \Gamma; v; H \vdash G)$ as $(\Gamma; v \vdash \{F\} hc \{\vdash G\})$.

Lemma 64 (Frame Lemma) *Let* $\Gamma \vdash H : \diamond$.

(a) *If* $(\Gamma; v \vdash \{F\} c : T \{(\texttt{ex } T'\,\alpha)\,(G)\})$ *and* $\mathsf{fv}(c) \cap \mathsf{fv}(H) \subseteq \mathsf{RdVar} \cup \mathsf{LogVar}$, *then* $(\Gamma; v \vdash \{F * H\} c : T \{(\texttt{ex } T'\,\alpha)\,(G * H)\})$.
(b) *If* $(\Gamma; v \vdash \{F\} hc \{G\})$ *and* $\mathsf{fv}(hc) \cap \mathsf{fv}(H) \subseteq \mathsf{RdVar} \cup \mathsf{LogVar}$, *then* $(\Gamma; v \vdash \{F * H\} hc \{\vdash G * H\})$.

Proof. By induction on $(\Gamma; v \vdash \{F\}c : T\{(\texttt{ex}\ T'\ \alpha)\ (G)\})$ and $(\Gamma; v \vdash \{F\}hc\{G\})$. □

Lemma 65 (Derived Rule for Bind) *If* $(\Gamma; o \vdash \{F\}c : T\{(\texttt{ex}\ T\ \alpha)\ (G)\})$, $T <: \Gamma(\ell)$ *and* $(\Gamma; p \vdash \{(\texttt{ex}\ T\ \alpha)\ (\alpha == \ell * G)\}c' : U\{H\})$, *then* $(\Gamma; o \vdash \{F\}\ell \leftarrow c; c' : U\{H\})$.

[Chris says: I defined the verification rule for return for object ids only. This is why these lemmas only hold for object ids, not for arbitrary values (including read-only variables and `null`). We, anyways, only need these lemmas for the case where the receiver parameter is an object id.]

Proof. By induction on the structure of c. □

Lemma 66 (Derived Rule for Sequential Composition) *If* $(\Gamma; o \vdash \{F\}c : \texttt{void}\{G\})$ *and* $(\Gamma; o \vdash \{G\}c' : T\{H\})$, *then* $(\Gamma; o \vdash \{F\}c; c' : T\{H\})$.

Proof. This is a consequence of the derived rule for bind (Lemma 65). □

Q Basic Properties of Semantics

Lemma 67 (Expression Semantics Preserves Typings) *If* $(\Gamma \vdash e : T)$, $(\Gamma_{\mathsf{hp}} \vdash h : \diamond)$, $(\Gamma \vdash s : \diamond)$ *and* $[\![e]\!]^h_s = \mu$, *then* $(\Gamma \vdash \mu : T)$.

Proof. By induction on $(\Gamma \vdash e : T)$. □

Lemma 68 (Pure Expression Have Values) *If* $(\Gamma \vdash e : T)$, $(\Gamma_{\mathsf{hp}} \vdash h : \diamond)$, $(\Gamma \vdash s : \diamond)$ *and* $\mathscr{Q}; h; s \models \mathsf{pure}(e)$, *then* $[\![e]\!]^h_s = \mu$ *for some some* μ.

Proof. By induction on $(\Gamma \vdash e : T)$. □

Lemma 69 (Stability of Pure Expressions) *If* $\mathscr{R} = (h, \mathscr{P}, \mathscr{Q})$, $h\#h'$, $(\mathscr{Q}; h; s \models \mathsf{pure}(e))$ *and* $[\![e]\!]^h_s = \mu$, *then* $(\mathscr{Q}; h'; s \models \mathsf{pure}(e))$ *and* $[\![e]\!]^{h'}_s = \mu$.

Proof. By induction on the structure of e, using the resource soundness condition (e) on $\mathscr{R}$. □

We define: An expression e is called a *field selection expression* iff it is of the form $e = e'.f$ for some e', f.

Lemma 70 (Pure Field Selections are Enough) *If* $(\mathscr{Q}; h; s \models \mathsf{pure}(e'))$ *for all field selection expressions* e' *that are subexpressions of* e, *then* $(\mathscr{Q}; h; s \models \mathsf{pure}(e))$.

Proof. By induction on the structure of e. The only proof case that makes use of the induction hypothesis is the case where e is of the form $e = op(\bar{e})$. □

Lemma 71 (Resource Monotonicity) (a) *If* $[\![e]\!]^h_s = \mu$ *and* $h \leq h'$, *then* $[\![e]\!]^{h'}_s = \mu$.
(b) *If* $\mathscr{Q}; h; s \models \mathsf{pure}(e)$ *and* $h \leq h'$, *then* $\mathscr{Q}; h'; s \models \mathsf{pure}(e)$.
(c) *If* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, $\mathscr{R} \leq \mathscr{R}'$, $\Gamma \subseteq_{\mathsf{hp}} \Gamma'$ *and* $(\Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond)$, *then* $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models F)$.

Proof. Part (a) by induction on the structure of e, using Lemma 14. Part (b) by induction on the structure of e. Part (c) by induction on the structure of F. For the cases where $F = o.P\langle\bar{\pi}\rangle$ or $F = o.P@C\langle\bar{\pi}\rangle$, one uses that predicate environments are monotone with respect to resources, by axiom (a). The most interesting proof cases are the ones where $F = F_1 * F_2$ and $F = F_1 \mathrel{-\!\!*} F_2$. So we do these in detail:

Let $F = F_1 * F_2$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, $\mathscr{R} \leq \mathscr{R}'$, $\Gamma \subseteq_{\mathsf{hp}} \Gamma'$ and $(\Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond)$. Then $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}_2; s \models F_2)$. Moreover, $\mathscr{R}' = \mathscr{R} * \mathscr{R}_3$ for some $\mathscr{R}_3$, by Lemma 21(b). By Lemma 26, we have $(\Gamma'_{\mathsf{hp}} \vdash (\mathscr{R}_1 \nearrow \mathscr{R}')_{\mathsf{hp}} : \diamond)$ and $(\Gamma'_{\mathsf{hp}} \vdash ((\mathscr{R}_2 * \mathscr{R}_3) \nearrow \mathscr{R}')_{\mathsf{hp}} : \diamond)$. Thus, by induction hypothesis, $(\Gamma' \vdash \mathscr{E}; \mathscr{R}_1 \nearrow \mathscr{R}'; s \models F_1)$ and

$(\Gamma' \vdash \mathscr{E}; (\mathscr{R}_2 * \mathscr{R}_3) \nearrow \mathscr{R}'; s \models F_2)$. Then $(\Gamma' \vdash \mathscr{E}; \mathscr{R}_1 \nearrow \mathscr{R}' * (\mathscr{R}_2 * \mathscr{R}_3) \nearrow \mathscr{R}'; s \models F)$. But $\mathscr{R}_1 \nearrow \mathscr{R}' * (\mathscr{R}_2 * \mathscr{R}_3) \nearrow \mathscr{R}' = \mathscr{R}'$, by Lemma 27.

Let $F = F_1 \mathrel{-\!\!*} F_2$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, $\mathscr{R} \leq \mathscr{R}'$, $\Gamma \subseteq_{\mathsf{hp}} \Gamma'$ and $(\Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond)$. Let $\Gamma_1 \supseteq_{\mathsf{hp}} \Gamma'$, $\mathscr{R} \# \mathscr{R}_1$ and $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1)$. By Lemma 21(b), we have $\mathscr{R}' = \mathscr{R} * \mathscr{R}_0$ for some $\mathscr{R}_0$. By Lemmas 26 and 10, we have $((\Gamma_1)_{\mathsf{hp}} \vdash ((\mathscr{R}_0 \nearrow \mathscr{R}_1) * \mathscr{R}_1)_{\mathsf{hp}} : \diamond)$. By Lemma 27, $(\mathscr{R}_0 \nearrow \mathscr{R}_1) * \mathscr{R}_1 = \mathscr{R}_0 * \mathscr{R}_1$, thus, $((\Gamma_1)_{\mathsf{hp}} \vdash (\mathscr{R}_0 * \mathscr{R}_1)_{\mathsf{hp}} : \diamond)$. Furthermore, $\mathscr{R}_1 \leq \mathscr{R}_0 * \mathscr{R}_1$, by Lemma 21(a). Therefore by induction hypothesis we obtain $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_0 * \mathscr{R}_1; s \models F_1)$. Because $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, we then get $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * (\mathscr{R}_0 * \mathscr{R}_1); s \models F_2)$. But $\mathscr{R} * \mathscr{R}_0 * \mathscr{R}_1 = \mathscr{R}' * \mathscr{R}_1$. □

Lemma 72 (Store Invariance) (a) *If* $s_{|\mathsf{fv}(e)} = s'_{|\mathsf{fv}(e)}$ *and* $[\![e]\!]^h_s = \mu$, *then* $[\![e]\!]^h_{s'} = \mu$.

(b) *If* $s_{|\mathsf{fv}(e)} = s'_{|\mathsf{fv}(e)}$ *and* $\mathscr{Q}; h; s \models \mathsf{pure}(e)$, *then* $\mathscr{Q}; h; s' \models \mathsf{pure}(e)$.

(c) *If* $s_{|\mathsf{fv}(F)} = s'_{|\mathsf{fv}(F)}$, $\Gamma_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$, $\Gamma_{|\mathsf{fv}(F)} = \Gamma'_{|\mathsf{fv}(F)}$, $(\Gamma' \vdash s' : \diamond)$ *and* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, *then* $(\Gamma' \vdash \mathscr{E}; \mathscr{R}; s' \models F)$.

Proof. Parts (a) and (b) by inductions on the structure of e, and part (c) by induction on the structure of F. The most interesting proof cases are for $F = F_1 \mathrel{-\!\!*} F_2$ and $F = (\mathtt{fa}\ T\ \alpha)(F')$. So we show these in detail.

Let $F = F_1 \mathrel{-\!\!*} F_2$, $s_{|\mathsf{fv}(F)} = s'_{|\mathsf{fv}(F)}$, $\Gamma_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$, $\Gamma_{|\mathsf{fv}(F)} = \Gamma'_{|\mathsf{fv}(F)}$, $(\Gamma' \vdash s' : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$. Let $\Gamma'_1 \supseteq_{\mathsf{hp}} \Gamma'$, $\mathscr{R} \# \mathscr{R}_1$ and $(\Gamma'_1 \vdash \mathscr{E}; \mathscr{R}_1; s' \models F_1)$. Define $\Gamma_1 = (\Gamma'_1)_{\mathsf{hp}} \cup \Gamma_{|\mathsf{Var}}$. Then $\Gamma_1 \supseteq_{\mathsf{hp}} \Gamma$. Furthermore, $(\Gamma_1)_{\mathsf{hp}} = (\Gamma'_1)_{\mathsf{hp}}$, $(\Gamma_1)_{|\mathsf{fv}(F)} = \Gamma_{|\mathsf{fv}(F)} = \Gamma'_{|\mathsf{fv}(F)} = (\Gamma'_1)_{|\mathsf{fv}(F)}$, and $(\Gamma_1 \vdash s : \diamond)$ by weakening $(\Gamma \vdash s : \diamond)$. Then by induction hypothesis, $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1)$. Because $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, we have $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models F_2)$. By induction hypothesis, $(\Gamma'_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s' \models F_2)$.

Let $F = (\mathtt{fa}\ T\ \alpha)(F')$. Let $s_{|\mathsf{fv}(F)} = s'_{|\mathsf{fv}(F)}$, $\Gamma_{\mathsf{hp}} = \Gamma'_{\mathsf{hp}}$, $\Gamma_{|\mathsf{fv}(F)} = \Gamma'_{|\mathsf{fv}(F)}$, $(\Gamma' \vdash s' : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$. Let $\Gamma'_0 \supseteq_{\mathsf{hp}} \Gamma'$, $\mathscr{R}_0 \geq \mathscr{R}$, $((\Gamma'_0)_{\mathsf{hp}} \vdash (\mathscr{R}'_0)_{\mathsf{hp}} : \diamond)$ and $((\Gamma'_0)_{\mathsf{hp}} \vdash \pi : T)$. Define $\Gamma_0 = (\Gamma'_0)_{\mathsf{hp}} \cup \Gamma_{|\mathsf{Var}}$ Then $\Gamma_0 \supseteq_{\mathsf{hp}} \Gamma$. Because $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, we have $(\Gamma_0 \vdash \mathscr{E}; \mathscr{R}_0; s \models F[\pi/\alpha])$. Then by induction hypothesis, $(\Gamma'_0 \vdash \mathscr{E}; \mathscr{R}_0; s' \models F[\pi/\alpha])$. □

Lemma 73 (Value Substitutivity for Semantics) (a) $[\![e[v/\ell]]\!]^h_s = \mu$ *iff* $[\![e]\!]^h_{s[\ell \mapsto v]} = \mu$.

(b) $(\mathscr{Q}; h; s \models \mathsf{pure}(e[v/\ell]))$ *iff* $(\mathscr{Q}; h; s[\ell \mapsto v] \models \mathsf{pure}(e))$.

(c) *If* $(\Gamma \vdash v : T)$, $(\Gamma \vdash s : \diamond)$ *and* $(\Gamma, \ell : T \vdash F : \diamond)$, *then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[v/\ell])$ *iff* $(\Gamma, \ell : T \vdash \mathscr{E}; \mathscr{R}; s[\ell \mapsto v] \models F)$.

Proof. Parts (a) and (b) by induction on the structure of e, and part (c) by induction on the structure of F. The most interesting proof cases are for $F = F_1 \mathrel{-\!\!*} F_2$ and $F = (\mathtt{fa}\ T\ \alpha)(F')$. So we show these in detail.

Let $F = F_1 \mathrel{-\!\!*} F_2$. Let $(\Gamma \vdash v : T)$, $(\Gamma \vdash s : \diamond)$, $(\Gamma, \ell : T \vdash F : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[v/\ell])$. Let $\Gamma_1 \supseteq_{\mathsf{hp}} (\Gamma, \ell : T)$, $\mathscr{R} \# \mathscr{R}_1$ and $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s[\ell \mapsto v] \models F_1)$. Define $\Gamma'_1 = \Gamma_1 \setminus \{\ell : T\}$. By induction hypothesis, we have $(\Gamma'_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1[v/\ell])$. Then $(\Gamma'_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models F_2[v/\ell])$, because $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[v/\ell])$. Then by induction hypothesis $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s[\ell \mapsto v] \models F_2)$.

Let $F = F_1 \mathrel{-\!\!*} F_2$. Let $(\Gamma \vdash v : T)$, $(\Gamma \vdash s : \diamond)$, $(\Gamma, \ell : T \vdash F : \diamond)$ and $(\Gamma, \ell : T \vdash \mathscr{E}; \mathscr{R}; s[\ell \mapsto v] \models F)$. Let $\Gamma_1 \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R} \# \mathscr{R}_1$ and $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1[v/\ell])$. By induction hypothesis, $(\Gamma_1, \ell : T \vdash \mathscr{E}; \mathscr{R}_1; s[\ell \mapsto v] \models F_1)$. Then $(\Gamma_1, \ell : T \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s[\ell \mapsto v] \models$

F_2), because $(\Gamma, \ell : T \vdash \mathscr{E}; \mathscr{R}; s[\ell \mapsto v] \models F)$. Then by induction hypothesis, $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models F_2[v/\ell])$.

Let $F = (\mathtt{fa}\, T\, \alpha)(G)$, $(\Gamma \vdash v : T)$, $(\Gamma \vdash s : \diamond)$, $(\Gamma, \ell : T \vdash F : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[v/\ell])$. Let $\Gamma' \supseteq_{\mathsf{hp}} (\Gamma, \ell : T)$, $\mathscr{R}' \geq \mathscr{R}$, $(\Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}} : \diamond)$ and $(\Gamma'_{\mathsf{hp}} \vdash \pi : T)$. Let $\Gamma'' = \Gamma' \setminus \{\ell : T\}$. Because $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[v/\ell])$, we have $(\Gamma'' \vdash \mathscr{E}; \mathscr{R}'; s \models G[v/\ell][\pi/\alpha])$. But $G[v/\ell][\pi/\alpha] = G[\pi/\alpha][v/\ell]$, because $\alpha \notin \mathsf{fv}(v)$ and π is closed. Therefore, $(\Gamma'' \vdash \mathscr{E}; \mathscr{R}'; s \models G[\pi/\alpha][v/\ell])$. Then by induction hypothesis, $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s[\ell \mapsto v] \models G[\pi/\alpha])$.

Let $F = (\mathtt{fa}\, T\, \alpha)(G)$, $(\Gamma \vdash v : T)$, $(\Gamma \vdash s : \diamond)$, $(\Gamma, \ell : T \vdash F : \diamond)$ and $(\Gamma, \ell : T \vdash \mathscr{E}; \mathscr{R}; s[\ell \mapsto v] \models F)$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}' \geq \mathscr{R}$, $(\Gamma'_{\mathsf{hp}} \vdash \mathscr{R}_{\mathsf{hp}} : \diamond)$ and $(\Gamma'_{\mathsf{hp}} \vdash \pi : T)$. Because $(\Gamma, \ell : T \vdash \mathscr{E}; \mathscr{R}; s[\ell \mapsto v] \models F)$, we have $(\Gamma', \ell : T \vdash \mathscr{E}; \mathscr{R}'; s[\ell \mapsto v] \models G[\pi/\alpha])$. By induction hypothesis, $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models G[\pi/\alpha][v/\ell])$. Then $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models G[v/\ell][\pi/\alpha])$, because $G[v/\ell][\pi/\alpha] = G[\pi/\alpha][v/\ell]$. □

Lemma 74 (Expression Substitutivity for Semantics) (a) *If* $[\![\pi_1]\!] = [\![\pi_2]\!]$, *then* $\pi[\pi_1/x] = \pi[\pi_2/x]$ *and* $T[\pi_1/x] = T[\pi_2/x]$.

(b) *If* $[\![e[e_1/x]]\!]^h_s = \mu$ *and* $[\![e_1]\!]^h_s = [\![e_2]\!]^h_s$, *then* $[\![e[e_2/x]]\!]^h_s = \mu$.

(c) *If* $(\mathscr{Q}; h; s \models \mathsf{pure}(e[e_1/x]))$, $[\![e_1]\!]^h_s = [\![e_2]\!]^h_s$ *and* $(\mathscr{Q}; h; s \models \mathsf{pure}(e_2))$, *then* $(\mathscr{Q}; h; s \models \mathsf{pure}(e[e_2/x]))$.

(d) *If* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_1/x])$, $h = \mathscr{R}_{\mathsf{hp}}$, $[\![e_1]\!]^h_s = [\![e_2]\!]^h_s$, $(\mathscr{R}_{\mathsf{glo}}; h; s \models \mathsf{pure}(e_1, e_2))$ *and* $(\Gamma \vdash F[e_2/x] : \diamond)$, *then* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_2/x])$.

Proof. Part (a) is an immediate consequence of the injectivity of value semantics (Lemma 5). Parts (b) and (c) are shown by inductions on the structure of e. Part (d) is shown by induction on the structure of F. To deal with specification parameters of predicates (resp., type annotations in quantifiers), one uses part (a) in combination with the fact that $\pi[e/x]$ (resp., $T[e/x]$) can only be well-typed if either x does not occur in π (resp., T) or e is a specification value π'. The most interesting proof case is for $F = F_1 \mathrel{-\!\!*} F_2$. So we show this case in detail:

Let $F = F_1 \mathrel{-\!\!*} F_2$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_1/x])$, $\mathscr{R} = (h, \mathscr{P}, \mathscr{Q})$, $[\![e_1]\!]^h_s = [\![e_2]\!]^h_s$ and $(\mathscr{Q}; h; s \models \mathsf{pure}(e_1, e_2))$ and $(\Gamma \vdash F[e_2/x] : \diamond)$. Let $\Gamma_1 \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}_1 = (h_1, \mathscr{P}_1, \mathscr{Q})$, $\mathscr{R} \# \mathscr{R}_1$ and $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1[e_2/x])$. By Lemma 69, we have $[\![e_1]\!]^{h_1}_s = [\![e_1]\!]^h_s = [\![e_2]\!]^h_s = [\![e_2]\!]^{h_1}_s$ and $\mathscr{Q}; h_1; s \models \mathsf{pure}(e_1, e_2)$. From $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_1/x])$ and weakening, it follows that $(\Gamma_1 \vdash F_1[e_1/x] : \diamond)$. Therefore, by induction hypothesis, $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models F_1[e_1/x])$. Then $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models F_2[e_1/x])$, because $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[e_1/x])$. Because $h \leq h * h_1$, we can apply resource monotonicity (Lemma 71) to get $[\![e_1]\!]^{h * h_1}_s = [\![e_1]\!]^h_s = [\![e_2]\!]^h_s = [\![e_2]\!]^{h * h_1}_s$ and $\mathscr{Q}; h * h_1; s \models \mathsf{pure}(e_1, e_2)$. Then $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models F_2[e_2/x])$, by induction hypothesis. □

In the following, recall that F^{sp} ranges over formulas that do not contain $\mathrel{-\!\!*}$, $|$, $\mathtt{fa}$ or predicate identifiers that are not data group identifiers.

Lemma 75 (Splitting) *If* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F^{\mathsf{sp}})$, *then* $(\Gamma \vdash \mathscr{E}; \frac{1}{2}\mathscr{R}; s \models \mathsf{split}(F^{\mathsf{sp}}))$.

Proof. By induction on the structure of F^{sp}. For cases $F^{\mathsf{sp}} = o.P^{\mathsf{grp}}@C\langle\bar{\pi}\rangle$ and $F^{\mathsf{sp}} = o.P^{\mathsf{grp}}\langle\bar{\pi}\rangle$, we use axiom (c) for predicate environments. For case $F^{\mathsf{sp}} = (\mathtt{ex}\, T\, \alpha)(G^{\mathsf{sp}})$, we use that splitting commutes with substitution: $\mathsf{split}(F^{\mathsf{sp}})[\pi/\alpha] = \mathsf{split}(F^{\mathsf{sp}}[\pi/\alpha])$. □

Lemma 76 (Merging Without Datagroups and Existentials) *If F^{sp} does not contain datagroup identifiers or existentials and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}}))$, then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F^{\mathsf{sp}})$.*

Proof. By induction on the structure of F^{sp}. □

The following lemma is needed to update the global permission table $\mathscr{Q}$ when a field $o.f$ gets `finalized`. We define:

$$\mathsf{finalize}(o.f, v, \mathscr{R}) \triangleq (\mathscr{R}_{\mathsf{hp}}[o.f \mapsto v], \mathscr{R}_{\mathsf{loc}}[(o,f) \mapsto 0], \mathscr{R}_{\mathsf{glo}}[(o,f) \mapsto 0])$$

Lemma 77 (Finalization) *Let $o \in \mathsf{dom}(\mathscr{R}_{\mathsf{hp}})$, $\mathscr{R}_{\mathsf{loc}}(o,f) = 0$, $\mathscr{R}_{\mathsf{glo}}(o,f) = 1$ and $(\Gamma_{\mathsf{hp}} \vdash \mathscr{R}_{\mathsf{hp}}[o.f \mapsto v] : \diamond)$.*

If $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, then $(\Gamma \vdash \mathscr{E}; \mathsf{finalize}(o.f, v, \mathscr{R}); s \models F)$.

Proof. By induction on the structure of F. For cases $F = o.P@C\langle\bar{\pi}\rangle$ and $F = o.P\langle\bar{\pi}\rangle$, we use axiom (e) for predicate environments. □

The following lemma is needed to update the global permission table after calling `join`.

Lemma 78 (Thread Joining) *Let $o \in \mathsf{dom}(h)$, $\mathscr{P}(o, \mathtt{join}) \leq x \leq \mathscr{Q}(o, \mathtt{join})$ and $\mathscr{Q}' = \mathscr{Q}[(o, \mathtt{join}) \mapsto x]$.*

If $(\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{Q}); s \models F)$, then $(\Gamma \vdash \mathscr{E}; (h, \mathscr{P}, \mathscr{Q}'); s \models F)$.

Proof. By induction on the structure of F. For cases $F = o.P@C\langle\bar{\pi}\rangle$ and $F = o.P\langle\bar{\pi}\rangle$, we use axiom (f) for predicate environments. □

The following lemma is needed to prove soundness of the verification rule (New).

$$\mathsf{initloc}(o)(p,k) \triangleq \begin{cases} 1 & \text{if } p = o \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{inithp}(\Gamma, o, T)(p) \triangleq \begin{cases} (T, \mathsf{init}(T)) & \text{if } p = o \\ (\Gamma(o), \emptyset) & \text{if } p \in \mathsf{dom}(\Gamma) \setminus \{o\} \end{cases}$$

$$\mathsf{initrsc}(\Gamma, o, T, \mathscr{Q}) \triangleq (\ \mathsf{inithp}(\Gamma, o, T),\ \mathsf{initloc}(o),\ \mathscr{Q}\ )$$

Lemma 79 (Initialization)

(a) *If $(\Gamma \vdash C\langle\bar{\pi}\rangle : \diamond)$, $o \notin \mathsf{dom}(\Gamma)$, $\mathscr{Q}(o,k) = 1$ for all k in $\mathsf{FieldId} \cup \{\mathtt{join}\}$, $(\Gamma \vdash s : \diamond)$, $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$ and $C \preceq D$, then $(\Gamma, o : C\langle\bar{\pi}\rangle \vdash \mathscr{E}; \mathsf{initrsc}(\Gamma, o, C\langle\bar{\pi}\rangle, \mathscr{Q}); s \models o.\mathtt{init@}D)$.*

(b) *If $(\Gamma \vdash C\langle\bar{\pi}\rangle : \diamond)$, $o \notin \mathsf{dom}(\Gamma)$, $\mathscr{Q}(o,k) = 1$ for all k in $\mathsf{FieldId} \cup \{\mathtt{join}\}$, $(\Gamma \vdash s : \diamond)$, and $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, then $(\Gamma, o : C\langle\bar{\pi}\rangle \vdash \mathscr{E}; \mathsf{initrsc}(\Gamma, o, C\langle\bar{\pi}\rangle, \mathscr{Q}); s \models o.\mathtt{init})$.*

Proof. Part (a) by induction on the subclassing order $\preceq$. Part (b) follows from part (a) because $o.\mathtt{init}$ is equivalent to $o.\mathtt{init@}C$, if C is o's dynamic class. □

R Soundness of Logical Consequence

Lemma 80 (Soundness of Syntactic Purity) *If* $(\mathcal{Q};h;s \models \bar{F} : \checkmark)$ *and* $(\bar{F} \vdash G : \checkmark)$, *then* $(\mathcal{Q};h;s \models G : \checkmark)$.

Proof. Let $(\mathcal{Q};h;s \models \bar{F} : \checkmark)$ and $(\bar{F} \vdash G : \checkmark)$. Let e be a field selection expression that occurs in G. Then e occurs in $\bar{F}$ by definition of $(\bar{F} \vdash G : \checkmark)$. Then $(\mathcal{Q};h;s \models \mathsf{pure}(e))$, because $(\mathcal{Q};h;s \models \bar{F} : \checkmark)$. □

We now prove soundness for the merge-restricted logical consequence judgment $\vdash'_{\mathsf{w}}$ (see Section G).

Lemma 81 (Soundness of Merge-restricted Logical Consequence) *If* $(\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G)$ *and* $\mathcal{F}_{ct}(\mathscr{E}) = \mathscr{E}$, *then* $(\Gamma \vdash \mathscr{E};\bar{F} \models G : \checkmark)$.

Proof. By induction on the height of the proof tree that we obtain from $(\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G)$'s proof tree after "inlining" proof trees for class axioms. This inlining is well-founded because proofs of class axiom are, by definition, not allowed to make use of class axioms (see Section J).

Case 1, (Id):

$$\frac{\Gamma;r \vdash \bar{F},G : \diamond}{\Gamma;r;\bar{F},G \vdash'_{\mathsf{w}} G}$$

Suppose $(\Gamma \vdash \mathscr{R},s : \diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F},G : \checkmark)$. By definition of semantic validity (case $\bar{F} * G$), there exists a $\mathscr{R}' \leq \mathscr{R}$ such that $(\Gamma \vdash \mathscr{E};\mathscr{R}';s \models G)$. Then $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models G)$, by resource monotonicity (Lemma 71). Moreover, $(\mathscr{R}_{\mathsf{glo}};\mathscr{R}_{\mathsf{hp}};s \models G : \checkmark)$ follows from $(\mathscr{R}_{\mathsf{glo}};\mathscr{R}_{\mathsf{hp}};s \models \bar{F},G : \checkmark)$, because every subexpression of G is a subexpression of $\bar{F},G$.

Case 2, (Pure Intro):

$$\frac{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G \quad \bar{F} \vdash e : \checkmark}{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G * \mathtt{Pure}(e)}$$

Let $(\Gamma \vdash \mathscr{R},s : \diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F} : \checkmark)$. Then $(\mathscr{R}_{\mathsf{glo}};\mathscr{R}_{\mathsf{hp}};s \models \bar{F} : \checkmark)$. Moreover, $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models G : \checkmark)$ by induction hypothesis. By Lemma 70, it is now enough to show that $\mathscr{R}_{\mathsf{glo}};\mathscr{R}_{\mathsf{hp}};s \models \mathsf{pure}(e')$ for all field selection subexpressions e' of e. So let e' be a field selection subexpression of e. Then e' occurs in $\bar{F}$, because $(\bar{F} \vdash e : \checkmark)$. Then $\mathscr{R}_{\mathsf{glo}};\mathscr{R}_{\mathsf{hp}};s \models \mathsf{pure}(e')$, because $(\mathscr{R}_{\mathsf{glo}};\mathscr{R}_{\mathsf{hp}};s \models \bar{F} : \checkmark)$.

Case 3, ($*$ Intro):

$$\frac{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} H_1 \quad \Gamma;r;\bar{G} \vdash'_{\mathsf{w}} H_2}{\Gamma;r;\bar{F},\bar{G} \vdash'_{\mathsf{w}} H_1 * H_2}$$

Let $(\Gamma \vdash \mathscr{R},s : \diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F},\bar{G} : \checkmark)$. By definition of semantic validity, there are $\mathscr{R}_1$ and $\mathscr{R}_2$ such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models \bar{F} : \checkmark)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R}_2;s \models \bar{G} : \checkmark)$. By induction hypothesis, $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models H_1 : \checkmark)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R}_2;s \models H_2 : \checkmark)$. Then, by definition of semantic validity, $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models H_1 * H_2 : \checkmark)$.

Case 4, (∗ Elim):

$$\frac{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 * G_2 \quad \Gamma; r; \bar{E}, G_1, G_2 \vdash'_{\mathsf{w}} H}{\Gamma; r; \bar{F}, \bar{E} \vdash'_{\mathsf{w}} H}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F}, \bar{E} : \checkmark)$. By definition of semantic validity, there are $\mathscr{R}_1$ and $\mathscr{R}_2$ such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}_1; s \models \bar{F} : \checkmark)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}_2; s \models \bar{E} : \checkmark)$. By induction hypothesis, $(\Gamma \vdash \mathscr{E}; \mathscr{R}_1; s \models G_1 * G_2 : \checkmark)$. Then, by definition of semantic validity, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{E}, G_1, G_2 : \checkmark)$. Then, by induction hypothesis, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models H : \checkmark)$.

Case 5, (−∗ Intro):

$$\frac{\Gamma; r; \bar{F}, G_1 \vdash'_{\mathsf{w}} G_2 \quad \bar{F} \vdash G_1 : \checkmark}{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 \mathrel{-\!\!*} G_2}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} : \checkmark)$. Note that $\mathscr{R}_{\mathsf{glo}}; \mathscr{R}_{\mathsf{hp}}; s \models G_1 : \checkmark$, by Lemma 80. Let $\Gamma_1 \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R} \# \mathscr{R}_1$ and $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R}_1; s \models G_1)$. By resource monotonicity (Lemma 71), we have $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} \nearrow \mathscr{R}_1; s \models \bar{F} : \checkmark)$. Furthermore, $\mathscr{R} * \mathscr{R}_1 = (\mathscr{R} \nearrow \mathscr{R}_1) * \mathscr{R}_1$, by Lemma 27. Therefore, $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models \bar{F}, G_1 : \checkmark)$. Then $(\Gamma_1 \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}_1; s \models G_2 : \checkmark)$, by induction hypothesis.

Case 6, (−∗ Elim):

$$\frac{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} H_1 \mathrel{-\!\!*} H_2 \quad \Gamma; r; \bar{G} \vdash'_{\mathsf{w}} H_1}{\Gamma; r; \bar{F}, \bar{G} \vdash'_{\mathsf{w}} H_2}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F}, \bar{G} : \checkmark)$. By definition of semantic validity, there are $\mathscr{R}_1$ and $\mathscr{R}_2$ such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}_1; s \models \bar{F} : \checkmark)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}_2; s \models \bar{G} : \checkmark)$. By induction hypothesis, $(\Gamma \vdash \mathscr{E}; \mathscr{R}_1; s \models H_1 \mathrel{-\!\!*} H_2 : \checkmark)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}_2; s \models H_1 : \checkmark)$. By definition of semantic validity, we then have $(\Gamma \vdash \mathscr{E}; \mathscr{R}_1 * \mathscr{R}_2; s \models H_2 : \checkmark)$.

Case 7, (& Intro):

$$\frac{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 \quad \Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_2}{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 \,\&\, G_2}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} : \checkmark)$. Then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_1 : \checkmark)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_2 : \checkmark)$, by induction hypothesis. Then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_1 \,\&\, G_2 : \checkmark)$, by definition of semantic validity.

Case 8, (& Elim 1):

$$\frac{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 \,\&\, G_2}{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} : \checkmark)$. Then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_1 \,\&\, G_2 : \checkmark)$, by induction hypothesis. Then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_1 : \checkmark)$, by definition of semantic validity.

Case 9, (| Intro 1):

$$\frac{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 \quad \bar{F} \vdash G_2 : \checkmark}{\Gamma; r; \bar{F} \vdash'_{\mathsf{w}} G_1 \mid G_2}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \bar{F} : \checkmark)$. Then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_1 : \checkmark)$, by definition of semantic validity. Then $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G_1 \mid G_2 : \checkmark)$, by definition of semantic validity and Lemma 80.

Case 10, (| Elim):

$$\frac{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G_1 \mid G_2 \quad \Gamma;r;\bar{E},G_1 \vdash'_{\mathsf{w}} H \quad \Gamma;r;\bar{E},G_2 \vdash'_{\mathsf{w}} H}{\Gamma;r;\bar{F},\bar{E} \vdash'_{\mathsf{w}} H}$$

Let $(\Gamma \vdash \mathscr{R},s:\diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F},\bar{E}:\checkmark)$. Then there are $\mathscr{R}_1$, $\mathscr{R}_2$ such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models \bar{F}:\checkmark)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R}_2;s \models \bar{E}:\checkmark)$. By induction hypothesis, $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models G_1 \mid G_2:\checkmark)$. This means that $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models G_1:\checkmark)$ or $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models G_2:\checkmark)$. Let's assume the former. Then $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{E},G_1:\checkmark)$. Then $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models H:\checkmark)$, by induction hypothesis. The other case is symmetric.

Case 11, (Ex Intro):

$$\frac{\Gamma \vdash \pi : T \quad \Gamma,\alpha:T \vdash G:\diamond \quad \Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G[\pi/\alpha]}{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} (\mathtt{ex}\ T\ \alpha)(G)}$$

Let $(\Gamma \vdash \mathscr{R},s:\diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F}:\checkmark)$. We know that $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$, by definition of $(\Gamma \vdash \mathscr{R},s:\diamond)$. Furthermore, π does not contain read-write variables, by definition of specification values. Thus $(\Gamma_{\mathsf{hp}} \vdash \pi : T)$, by strengthening $(\Gamma \vdash \pi : T)$. Moreover $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models G[\pi/\alpha]:\checkmark)$, by induction hypothesis. Then $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models (\mathtt{ex}\ T\ \alpha)(G))$, by definition of semantic validity. To obtain $(\mathscr{R}_{\mathsf{hp}};\mathscr{R}_{\mathsf{glo}};s \models (\mathtt{ex}\ T\ \alpha)(G):\checkmark)$ we note that every field selection expression that occurs in $(\mathtt{ex}\ T\ \alpha)(G)$ also occurs in $G[\pi/\alpha]$, by our syntactic restriction that the bound variable α must not occur in field selection expressions.

Case 12, (Ex Elim):

$$\frac{\Gamma;r;\bar{E} \vdash'_{\mathsf{w}} (\mathtt{ex}\ T\ \alpha)(G) \quad \Gamma,\alpha:T;r;\bar{F},G \vdash'_{\mathsf{w}} H \quad \alpha \notin \bar{F},H}{\Gamma;r;\bar{E},\bar{F} \vdash'_{\mathsf{w}} H}$$

Let $(\Gamma \vdash \mathscr{R},s:\diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{E},\bar{F}:\checkmark)$. Then there are $\mathscr{R}_1$ and $\mathscr{R}_2$ such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models \bar{E}:\checkmark)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R}_2;s \models \bar{F}:\checkmark)$. Then $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models (\mathtt{ex}\ T\ \alpha)(G):\checkmark)$, by induction hypothesis. Then there exists some π such that $(\Gamma_{\mathsf{hp}} \vdash \pi : T)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R}_1;s \models G[\pi/\alpha]:\checkmark)$. Then $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{E},G[\pi/\alpha]:\checkmark)$. On the other hand, we have $(\Gamma;r;\bar{E},G[\pi/\alpha] \vdash'_{\mathsf{w}} H)$, by substitutivity (Lemma 49). Because value substitutions do not increase derivation height, we can apply the induction hypothesis to obtain $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models H:\checkmark)$.

Case 13, (Fa Intro):

$$\frac{\alpha \notin \bar{F} \quad \Gamma,\alpha:T;r;\bar{F} \vdash'_{\mathsf{w}} G}{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} (\mathtt{fa}\ T\ \alpha)(G)}$$

Let $(\Gamma \vdash \mathscr{R},s:\diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F}:\checkmark)$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}' \geq \mathscr{R}$, $(\Gamma'_{\mathsf{hp}} \vdash \mathscr{R}'_{\mathsf{hp}}:\diamond)$ and $(\Gamma'_{\mathsf{hp}} \vdash \pi : T)$. By resource monotonicity (Lemma 71), we have $(\Gamma' \vdash \mathscr{E};\mathscr{R}';s \models \bar{F}:\checkmark)$. By weakening (Lemma 47), we have $(\Gamma',\alpha:T;r;\bar{F} \vdash'_{\mathsf{w}} G)$. Then $(\Gamma';r;\bar{F} \vdash'_{\mathsf{w}} G[\pi/\alpha])$, by $\alpha \notin \bar{F}$ and substitutivity (Lemma 49). Now we can apply the induction hypothesis to obtain $(\Gamma \vdash \mathscr{E};\mathscr{R}';s \models G:\checkmark)$.

Case 14, (Fa Elim):

$$\frac{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} (\mathtt{fa}\ T\ \alpha)(G) \quad \Gamma \vdash \pi : T}{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G[\pi/\alpha]}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F} : \checkmark)$. By induction hypothesis, we have $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models (\mathtt{fa}\ T\ \alpha)(G) : \checkmark)$. Because $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$ and specification values do not contain read-write variables, we can strengthen $(\Gamma \vdash \pi : T)$ to obtain $(\Gamma_{\mathsf{hp}} \vdash \pi : T)$. Therefore, $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models G[\pi/\alpha])$ by the semantics of universal quantification. Moreover, $(\mathscr{R}_{\mathsf{hp}};\mathscr{R}_{\mathsf{glo}};s \models G[\pi/\alpha] : \checkmark)$ follows from $(\mathscr{R}_{\mathsf{hp}};\mathscr{R}_{\mathsf{glo}};s \models (\mathtt{fa}\ T\ \alpha)(G) : \checkmark)$ because α is not contained in field selection expressions, by syntactic restriction.

Case 15, (Ax):

$$\frac{\Gamma;r \vdash'_{\mathsf{w}} G \quad \Gamma \vdash \bar{F}, G : \diamond \quad \bar{F} \vdash G : \checkmark}{\Gamma;r;\bar{F} \vdash'_{\mathsf{w}} G}$$

Let $(\Gamma \vdash \mathscr{R}, s : \diamond)$ and $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \bar{F} : \checkmark)$. Note that $(\mathscr{R}_{\mathsf{hp}};\mathscr{R}_{\mathsf{glo}};s \models G : \checkmark)$ holds by Lemma 80. This takes care of the checkmark $\checkmark$, and it suffices to show that $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models G)$. To this end, we distinguish cases axiom by axiom:

Case 15.1:

$$\Gamma;r \vdash'_{\mathsf{w}} \mathtt{true}$$

Let $h = \mathscr{R}_{\mathsf{hp}}$. We have $([\![\mathtt{true}]\!]^h_s = \mathtt{true})$. Therefore, $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \mathtt{true} : \checkmark)$, by definition of semantic validity.

Case 15.2:

$$\Gamma;r \vdash'_{\mathsf{w}} \mathtt{false} \mathrel{-\!\!*} F$$

Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E};\mathscr{R}';s \models \mathtt{false})$. Let $h' = \mathscr{R}'_{\mathsf{hp}}$. Then $[\![\mathtt{false}]\!]^{h'}_s = \mathtt{true}$. This is impossible, so $(\Gamma \vdash \mathscr{E};\mathscr{R};s \models \mathtt{false} \mathrel{-\!\!*} F)$ is vacuously true.

Case 15.3:

$$\Gamma;r \vdash'_{\mathsf{w}} F^{\mathsf{sp}} \mathrel{-\!\!*} (\mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}}))$$

Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E};\mathscr{R}';s \models F^{\mathsf{sp}})$. Then $(\Gamma' \vdash \mathscr{E};\mathscr{R} * \mathscr{R}';s \models F^{\mathsf{sp}})$, by resource monotonicity (Lemma 71). Let $\mathscr{R} * \mathscr{R}' = (h, \mathscr{P}, \mathscr{Q})$. By Lemma 75, we obtain $(\Gamma' \vdash \mathscr{E};(h, \frac{1}{2}\mathscr{P}, \mathscr{Q});s \models \mathsf{split}(F^{\mathsf{sp}}))$. Because $(h, \frac{1}{2}\mathscr{P}, \mathscr{Q}) * (h, \frac{1}{2}\mathscr{P}, \mathscr{Q}) = (h, \mathscr{P}, \mathscr{Q})$, we then get $(\Gamma' \vdash \mathscr{E};(h, \mathscr{P}, \mathscr{Q});s \models \mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}}))$. Because $(h, \mathscr{P}, \mathscr{Q}) = \mathscr{R} * \mathscr{R}'$, we then have $(\Gamma' \vdash \mathscr{E};\mathscr{R} * \mathscr{R}';s \models \mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}}))$.

Case 15.4:

$$F^{\mathsf{sp}} \text{ does not contain datagroup ids} \ \Rightarrow\ \Gamma;r \vdash'_{\mathsf{w}} (\mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}})) \mathrel{-\!\!*} F^{\mathsf{sp}}$$

Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E};\mathscr{R}';s \models \mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}}))$. Then we have $(\Gamma' \vdash \mathscr{E};\mathscr{R} * \mathscr{R}';s \models \mathsf{split}(F^{\mathsf{sp}}) * \mathsf{split}(F^{\mathsf{sp}}))$, by resource monotonicity (Lemma 71). By Lemma 76, we then obtain $(\Gamma' \vdash \mathscr{E};\mathscr{R} * \mathscr{R}';s \models F^{\mathsf{sp}})$.

Case 15.5:

$$\Gamma;r \vdash'_{\mathsf{w}} (\mathtt{PointsTo}(e[f], \pi, e')\ \&\ \mathtt{PointsTo}(e[f], \pi', e''))\ \mathtt{assures}\ e' == e''$$

This axioms holds because heaps are functional in object- and field identifiers.

Case 15.6:

$$(\Gamma \vdash e : T) \;\Rightarrow\; \Gamma; r \vdash'_{\mathsf{w}} \texttt{Pure(}e\texttt{) -* (ex }T\ \alpha\texttt{)(}e\texttt{ == }\alpha\texttt{)}$$

This is a consequence of Lemma 68.

Case 15.7:

$$(\Gamma \models \texttt{!}e_1 \mid \texttt{!}e_2 \mid e') \;\Rightarrow\; \Gamma; r \vdash'_{\mathsf{w}} \texttt{(}e_1 * e_2\texttt{) -* }e'$$

Let $(\Gamma \models \texttt{!}e_1 \mid \texttt{!}e_2 \mid e')$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models e_1 * e_2)$. Let $h = \mathscr{R}_{\mathsf{hp}}$, $h' = \mathscr{R}'_{\mathsf{hp}}$ and h'' be some total heap such that $(\Gamma'_{\mathsf{hp}} \vdash h'' : \diamond)$ and $h * h' \leq h''$. We have $[\![e_1]\!]^{h'}_s = [\![e_2]\!]^{h'}_s = \texttt{true}$, by definition of semantic validity. Then $[\![e_1]\!]^{h''}_s = [\![e_2]\!]^{h''}_s = \texttt{true}$, by Lemma 71. Then $[\![e']\!]^{h''}_s = \texttt{true}$, because $(\Gamma \models \texttt{!}e_1 \mid \texttt{!}e_2 \mid e')$. Furthermore, we have $(\mathscr{R}_{\mathsf{glo}}; h; s \models \mathsf{pure}(\texttt{!}e_1 \mid \texttt{!}e_2 \mid e'))$, thus, $(\mathscr{R}_{\mathsf{glo}}; h; s \models \mathsf{pure}(e'))$, thus, $(\mathscr{R}_{\mathsf{glo}}; h * h'; s \models \mathsf{pure}(e'))$ by Lemma 71. Then $[\![e']\!]^{h*h'}_s = \mu$ for some μ, by Lemma 68. Then $\texttt{true} = [\![e']\!]^{h''}_s = \mu$ by Lemma 71. Thus, $[\![e']\!]^{h*h'}_s = \texttt{true}$.

Case 15.8:

$$(\Gamma \vdash e, e' : T \;\wedge\; \Gamma, x : T \vdash F : \diamond) \Rightarrow \Gamma; r \vdash'_{\mathsf{w}} \texttt{(}F[e/x] * e \texttt{ == } e'\texttt{) -* }F[e'/x]$$

Let $(\Gamma \vdash e, e' : T)$ and $(\Gamma, x : T \vdash F : \diamond)$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models F[e/x] * e == e')$. Then $(\Gamma' \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models F[e/x] * e == e')$, by resource monotonicity (Lemma 71). Let $\mathscr{R} * \mathscr{R}' = (h, \mathscr{P}, \mathscr{Q})$. By definition of semantic validity, we have $(\Gamma' \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models F[e/x])$, $(\mathscr{Q}; h; s \models \mathsf{pure}(e, e'))$ and $[\![e]\!]^h_s = [\![e']\!]^h_s$. By substitutivity (Lemma 42) and weakening, we have $(\Gamma' \vdash F[e'/x] : \diamond)$. Thus, we can apply Lemma 74 to obtain $(\Gamma' \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models F[e'/x])$.

Case 15.9:

$$(\Gamma \vdash \pi : T \;\wedge\; \mathsf{axiom}(T) = G') \Rightarrow \Gamma; r \vdash'_{\mathsf{w}} G'[\pi/\texttt{this}]$$

Let $(\Gamma \vdash \pi : T)$ and $\mathsf{axiom}(T) = G'$. We need to show $(\Gamma \vdash \mathscr{E}; \mathscr{R}; r \models s = G'[\pi/\texttt{this}])$. Because $\mathsf{axiom}(T)$ is defined, T must be a reference type. Then π must be an object identifier. Let $\mathscr{R}_{\mathsf{hp}}(\pi)_1 = C\texttt{<}\overline{\pi'}\texttt{>}$. Let $|\overline{\alpha}| = |\overline{\pi'}|$. Let $G'' = \mathsf{axiom}(C\texttt{<}\overline{\alpha}\texttt{>})$. Because $C\texttt{<}\overline{\pi'}\texttt{>} <: T$, we know that $G''[\overline{\pi'}/\overline{\alpha}]$ implies G', and it therefore suffices to show that $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models G''[\overline{\pi'}, \pi/\overline{\alpha}, \texttt{this}])$. To this end, it suffices to show $\Gamma \vdash \mathscr{E}; C\ \texttt{isclassof}\ \pi \models G''[\overline{\pi'}, \pi/\overline{\alpha}, \texttt{this}]$. By soundness of class axioms (see Section J), we know that $\overline{\alpha} : \overline{T}, \texttt{this} : C\texttt{<}\overline{\alpha}\texttt{>}; \texttt{this}; C\ \texttt{isclassof this} \vdash'' G''$. But then we have $(\Gamma; \pi; C\ \texttt{isclassof}\ \pi \vdash'' G''[\overline{\pi'}, \pi/\overline{\alpha}, \texttt{this}])$, by substitutivity. Then $\Gamma \vdash \mathscr{E}; \texttt{true} \models G''[\overline{\pi'}, \pi/\overline{\alpha}, \texttt{this}]$, by induction hypothesis.

Case 15.10:

$$\Gamma; r \vdash'_{\mathsf{w}} \pi.P\texttt{@Object}$$

We need to show that $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \pi.P\texttt{@Object})$. Because this is true only if $\Gamma \vdash \pi.P\texttt{@Object} : \diamond$, we know that π is either `null` or an object identifier. In the former case, we are done by the semantics of predicates with `null`-receiver. So let $\pi = o$. By the semantics of predicates, we need to show that $\mathscr{E}(P\texttt{@Object})((), \mathscr{R}, o, ()) = 1$. This expression is only well-typed if $P \in \{\texttt{state}, \texttt{init}\}$. Because $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, we know that $\mathscr{F}_{ct}(\mathscr{E})(P\texttt{@Object})((), \mathscr{R}, o, ()) = 1$. By definition of $\mathscr{F}$, this is the case only if $(\Gamma \vdash \mathscr{E}; \mathscr{R}; \emptyset \models \texttt{true} * \texttt{true})$. But this is true.

Case 15.11:

$$\Gamma; r \vdash'_{\mathsf{w}} \texttt{null.}\kappa\texttt{<}\bar{\pi}\texttt{>}$$

$(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \texttt{null.}\kappa\texttt{<}\bar{\pi}\texttt{>})$ is true by the semantics of predicates with `null`-receiver.

Case 15.12:

$$C \preceq D \;\Rightarrow\; \Gamma; r \vdash'_{\mathsf{w}} \pi.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{@}C\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>}$$

If $\pi = \texttt{null}$, then this amounts to `true ispartof true`, which trivially holds. So let's assume that $\pi = o$ for some object identifier o. To abbreviate, let's write P_c for $o.P\texttt{@}C\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>}$ and P_d for $o.P\texttt{@}D\texttt{<}\bar{\pi}\texttt{>}$. By transitivity of `ispartof` (Lemma 53) and because we have already shown the soundness of the natural deduction rules for $*$ and $-\!\!*$, we can assume that D is an immediate superclass of C. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models P_c)$. Then $(\Gamma \vdash o : C\texttt{<}\bar{\pi}''\texttt{>})$ and $\mathscr{E}(P\texttt{@}C)(\bar{\pi}'', \mathscr{R}', o, (\bar{\pi}, \bar{\pi}')) = 1$. Because $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, then $\mathscr{F}_{ct}(\mathscr{E})(P\texttt{@}C)(\bar{\pi}'', \mathscr{R}', o, (\bar{\pi}, \bar{\pi}')) = 1$. By definition of $\mathscr{F}$, it follows that $\mathsf{pbody}(o.P\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>}, C\texttt{<}\bar{\pi}''\texttt{>}) = F\ \texttt{ext}\ D\texttt{<}\bar{\pi}'''\texttt{>}$ and $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models F * P_d)$. Then $\mathscr{R}' = \mathscr{R}'_1 * \mathscr{R}'_2$, $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'_1; s \models F)$ and $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'_2; s \models P_d)$ for some $\mathscr{R}'_1, \mathscr{R}'_2$. We need to show that $(\Gamma' \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models P_d * (P_d \mathrel{-\!\!*} P_c))$. To this end, it suffices to show that $(\Gamma' \vdash \mathscr{E}; \mathscr{R} \nearrow \mathscr{R}'; s \models F \mathrel{-\!\!*} P_d \mathrel{-\!\!*} P_c)$. To abbreviate, let $\mathscr{R}_0 = \mathscr{R} \nearrow \mathscr{R}'$. Using the natural deduction rules for $*$ and $-\!\!*$, we can derive $F \mathrel{-\!\!*} P_d \mathrel{-\!\!*} P_c$ from $F * P_d \mathrel{-\!\!*} P_c$. Because we have already shown the soundness of these natural deduction rules it suffices to show that $(\Gamma' \vdash \mathscr{E}; \mathscr{R}_0; s \models F * P_d \mathrel{-\!\!*} P_c)$. So let $\Gamma'' \supseteq_{\mathsf{hp}} \Gamma'$, $\mathscr{R}_0\#\mathscr{R}''$ and $(\Gamma'' \vdash \mathscr{E}; \mathscr{R}''; s \models F * P_d)$. By definition of predicate semantics and because $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$ it then follows that $(\Gamma'' \vdash \mathscr{E}; \mathscr{R}''; s \models P_c)$. Then $(\Gamma'' \vdash \mathscr{E}; \mathscr{R}_0 * \mathscr{R}''; s \models P_c)$, by resource monotonicity (Lemma 71).

Case 15.13:

$$\Gamma; r \vdash'_{\mathsf{w}} \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{ispartof}\ \pi.P\texttt{<}\bar{\pi}\texttt{>}$$

If $\pi = \texttt{null}$, then this amounts to `true ispartof true`, which trivially holds. So let's assume that $\pi = o$ for some object identifier o. To abbreviate, let's write P_c for $o.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>}$ and P for $o.P\texttt{<}\bar{\pi}\texttt{>}$. Let $\Gamma' \supseteq_{\mathsf{hp}} \Gamma$, $\mathscr{R}\#\mathscr{R}'$ and $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models P)$. Let $\mathscr{R}_{\mathsf{hp}}(o)_1 = D\texttt{<}\bar{\pi}''\texttt{>}$. Then $D \preceq C$, by well-typedness of P_c. By the semantics of predicates, there are $\bar{\pi}'$ such that $(\Gamma' \vdash \mathscr{E}; \mathscr{R}'; s \models o.P\texttt{@}D\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>})$. To abbreviate, let's write P_d for $o.P\texttt{@}D\texttt{<}\bar{\pi},\bar{\pi}'\texttt{>}$. By the previous axiom (which is already proven sound), we know that $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models P_c\ \texttt{ispartof}\ P_d)$. Therefore, $(\Gamma' \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models P_c * (P_c \mathrel{-\!\!*} P_d))$. It follows that $(\Gamma' \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models P_d)$. But then $(\Gamma \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models P)$, by predicate semantics.

Case 15.14:

$$\Gamma; r \vdash'_{\mathsf{w}} \pi.P\texttt{<}\bar{\pi}\texttt{>} \mathrel{*\!\!-\!\!*} (\texttt{ex}\ \bar{T}\,\bar{\alpha})(\pi.P\texttt{<}\bar{\pi},\bar{\alpha}\texttt{>})$$

It is straightforward to show the two implications. (Recall that the semantics of predicates with missing arguments looks up the predicate type in the dynamic class of the receiver and existentially quantifies over the missing parameters.)

Case 15.15:

$$\Gamma; r \vdash'_{\mathsf{w}} (\ \pi.P\texttt{@}C\texttt{<}\bar{\pi}\texttt{>} * C\ \texttt{isclassof}\ \pi\ ) \mathrel{-\!\!*} \pi.P\texttt{<}\bar{\pi}\texttt{>}$$

Proof straightforward.

Case 15.16:

$$(C \text{ is } \texttt{final} \text{ or } P \text{ is } \texttt{final} \text{ in } C) \;\Rightarrow\; \Gamma; r \vdash'_{\mathsf{w}} \pi.P@C\langle\bar{\pi}\rangle \mathrel{-\!\!*} \pi.P\langle\bar{\pi}\rangle$$

Proof straightforward.

Case 15.17:

$$\begin{array}{l}(\Gamma \vdash r : C\langle\bar{\pi}''\rangle \wedge \mathsf{pbody}(r.P\langle\bar{\pi},\bar{\pi}'\rangle, C\langle\bar{\pi}''\rangle) = F \text{ ext } D\langle\bar{\pi}'''\rangle) \\ \quad\Rightarrow\; \Gamma; r \vdash'_{\mathsf{w}} r.P@C\langle\bar{\pi},\bar{\pi}'\rangle \mathrel{*\!\!-\!\!*} (F * r.P@D\langle\bar{\pi}\rangle)\end{array}$$

Proof straightforward, using that $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$ and inspecting the definition of $\mathscr{F}$. □

In order to lift the soundness lemma from the merge-restricted logical consequence $\vdash'_{\mathsf{w}}$ to $\vdash$, we need to show soundness of the unrestricted merging axiom.

Lemma 82 (Merging) *If* $\Gamma; r \vdash_{\mathsf{w}} \mathsf{split}(F) : \mathsf{supp}$ *and* $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, *then* $\Gamma \vdash \mathscr{E}; \mathsf{split}(F) * \mathsf{split}(F) \models F$.

Proof. By induction on the structure of F. The case $F = o.P^{\mathsf{grp}}@C\langle\bar{\pi}\rangle$ uses axiom (d) for predicate environments. The case $F = o.P^{\mathsf{grp}}\langle\bar{\pi}\rangle$ also uses axiom (d) together with the fact that the existential quantifier in the semantics of $o.P^{\mathsf{grp}}\langle\bar{\pi}\rangle$ is empty, because datagroups do not have variable arity (as enforced by the subtyping rule (Grp Sub)). The only other interesting case is the one for existentials:

$$\frac{\Gamma \vdash T : \diamond \quad \Gamma, \alpha : T; r \vdash F : \mathsf{supp} \quad \Gamma, \alpha : T, \alpha' : T; r; \texttt{true} \vdash'_{\mathsf{w}} \mathsf{split}(F \,\&\, F[\alpha'/\alpha]) \mathrel{-\!\!*} \alpha == \alpha'}{\Gamma; r \vdash (\texttt{ex}\ T\ \alpha)(\mathsf{split}(F)) : \mathsf{supp}}$$

Let $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \mathsf{split}((\texttt{ex}\ T\ \alpha)(F)) * \mathsf{split}((\texttt{ex}\ T\ \alpha)(F)))$. Then there are $\mathscr{R}_1$, $\mathscr{R}_2$, π_1, π_2 such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}_i; s \models \mathsf{split}(F[\pi_i/\alpha]))$ for $i = 1, 2$. Using resource monotonicity, we obtain $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \mathsf{split}(F[\pi_1/\alpha] \,\&\, F[\pi_2/\alpha]))$. By applying substitutivity to the last premise of the above rule, we obtain $\Gamma; r; \texttt{true} \vdash'_{\mathsf{w}} \mathsf{split}(F[\pi_1/\alpha] \,\&\, F[\pi_2/\alpha]) \mathrel{-\!\!*} \pi_1 == \pi_2$. We now use the soundness of $\vdash'_{\mathsf{w}}$ (Lemma 81) to obtain $\Gamma \vdash \mathscr{E}; \mathsf{final}(\mathscr{R}); s \models \mathsf{split}(F[\pi_1/\alpha] \,\&\, F[\pi_2/\alpha]) \mathrel{-\!\!*} \pi_1 == \pi_2$. Thus, $\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \pi_1 == \pi_2$, by the semantics of linear implication and because $\mathsf{final}(\mathscr{R}) * \mathscr{R} = \mathscr{R}$. This means that $[\![\pi_1]\!] = [\![\pi_2]\!]$, hence, $\pi_1 = \pi_2$ (Lemma 5). Then we have $\mathsf{split}(F[\pi_1/\alpha]) = \mathsf{split}(F[\pi_2/\alpha])$, hence, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F[\pi_1/\alpha])$ by induction hypothesis. By the semantics of existentials, it follows that $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models (\texttt{ex}\ T\ \alpha)(F))$. □

Proof of Theorem 4 (Soundness of Logical Consequence). *If* $(\Gamma; r; \bar{F} \vdash G)$ *and* $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, *then* $(\Gamma \vdash \mathscr{E}; \bar{F} \models G : \checkmark)$.

Proof. By induction on the height of the derivation of $(\Gamma; r; \bar{F} \vdash G)$. All proof cases are exactly like in the proof of Lemma 81. The only additional axiom that needs to be shown is the merging axiom "$\Gamma; v \vdash F : \mathsf{supp} \;\Rightarrow\; \Gamma; v \vdash (\mathsf{split}(F) * \mathsf{split}(F)) \mathrel{-\!\!*} F$". But this is a consequence of Lemma 82 and the fact that $(\Gamma; v \vdash F : \mathsf{supp})$ implies $(\Gamma; v \vdash_{\mathsf{w}} F : \mathsf{supp})$. □

S The Formula Support

Our Hoare rule for `join` allows callers to use fractions of `join`'s post-condition. In order to prove the soundness of the `join` rule, we need the following property: *if* $(\Gamma \vdash \mathscr{E}; F \models G)$, *then* $(\Gamma \vdash \mathscr{E}; fr \cdot F \models fr \cdot G)$. Specifically, this property is needed to deal with subtyping: If F is `run`'s postcondition at the receiver's dynamic type and G the postcondition at its static type, we know that $(\Gamma \vdash \mathscr{E}; F \models G)$, by behavioral subtyping. What we need to know is $(\Gamma \vdash \mathscr{E}; fr \cdot F \models fr \cdot G)$, where fr is the fraction of the postcondition that the caller "requests". Provided that F and G are supported (which is a requirement for `run`'s postcondition), this property follows from a similar property for split: *if* $(\Gamma \vdash \mathscr{E}; F \models G)$, *then* $(\Gamma \vdash \mathscr{E}; \mathsf{split}(F) \models \mathsf{split}(F))$.

As a technical tool for proving this property, we define a function that maps formulas to their support:

$$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(e) \triangleq \mathscr{R} \qquad \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(\texttt{Pure(}e\texttt{)}) \triangleq \mathscr{R} \qquad \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(\texttt{null.}\kappa\texttt{<}\overline{\pi}\texttt{>}) \triangleq \mathscr{R}$$

$$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(o.P^{\mathsf{grp}}\texttt{@}C\texttt{<}\overline{\pi}\texttt{>}) \triangleq \bigwedge\{\mathscr{R}' \geq \mathscr{R} \mid (\exists \overline{\pi'})(\Gamma \vdash o : C\texttt{<}\overline{\pi'}\texttt{>},\ \mathscr{E}(P\texttt{@}C)(\overline{\pi'}, \mathscr{R}', o, \overline{\pi}) = 1)\}$$

$$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(o.P^{\mathsf{grp}}\texttt{<}\overline{\pi}\texttt{>}) \triangleq \bigwedge\{\mathscr{R}' \geq \mathscr{R} \mid (\exists \overline{\pi'})(\mathscr{R}_{\mathsf{hp}}(o)_1 = C\texttt{<}\overline{\pi'}\texttt{>},\ \mathscr{E}(P\texttt{@}C)(\overline{\pi'}, \mathscr{R}', o, \overline{\pi}) = 1)\}$$

$$\chi_{\Gamma,\mathscr{E},(h,\mathscr{P},\mathscr{Q}),s}(\texttt{PointsTo(}e\texttt{[}f\texttt{]},\pi,e'\texttt{)}) \triangleq \begin{cases} (h \cup \{(o,(T,(f,v)))\},\ \mathbf{0}[(o,f) \mapsto [\![\pi]\!]],\ \mathscr{Q}) \\ \text{where } [\![e]\!]^h_s = o,\ h(o)_1 = T,\ [\![e']\!]^h_s = v \end{cases}$$

$$\chi_{\Gamma,\mathscr{E},(h,\mathscr{P},\mathscr{Q}),s}(\texttt{Perm(}e\texttt{[join]},\pi\texttt{)}) \triangleq (h,\ \mathbf{0}[(o,\texttt{join}) \mapsto [\![\pi]\!]],\ \mathscr{Q})$$

$$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(F * G) \triangleq \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(F) * \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(G)$$

$$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(F \,\&\, G) \triangleq \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(F) \vee \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(G)$$

$$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(\texttt{(ex}\ T\ \alpha\texttt{)(}F\texttt{)}) \triangleq \begin{cases} \chi_{\Gamma,\mathscr{E},\mathscr{R},s}(F[\pi/\alpha]) \text{ where } \pi \text{ is unique such that} \\ \Gamma_{\mathsf{hp}} \vdash \pi : T \text{ and } (\exists \mathscr{R}' \geq \mathscr{R})(\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models (F[\pi/\alpha])) \end{cases}$$

$\chi_{\Gamma,\mathscr{E},\mathscr{R},s}(F)$ is not always well-defined, not even if $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$. For instance, the right-hand-sides of the clauses for `PointsTo` and `Perm` may not be well-formed resources, and the witness in the clause for existentials may not be unique. This is unproblematic, because we will only apply this function to resources of the form $\mathsf{final}(\mathscr{R})$ and to formulas F that are supported.

Lemma 83 (Existence and Minimality) *Let* $(\Gamma; o \vdash_{\mathsf{w}} F : \mathsf{supp})$, $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, *and* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$. *Then:*

(a) $\chi_{\Gamma,\mathscr{E},\mathsf{final}(\mathscr{R}),s}(F)$ *is defined.*
(b) $\Gamma \vdash \mathscr{E}; \chi_{\Gamma,\mathscr{E},\mathsf{final}(\mathscr{R}),s}(F); s \models F$.
(c) *If* $\mathscr{R}' \geq \mathsf{final}(\mathscr{R})$ *and* $\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models F$, *then* $\mathscr{R}' \geq \chi_{\Gamma,\mathscr{E},\mathsf{final}(\mathscr{R}),s}(F)$.

Proof. By induction on the structure of F. The proof uses axiom (b) for predicate environments. □

Lemma 84 (Splitting for Semantic Entailment) *If* $(\Gamma; o \vdash_{\mathsf{w}} F, G, \mathsf{split}(F), \mathsf{split}(G) : \mathsf{supp})$ *and* $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$ *and* $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$ *and* $(\Gamma \vdash \mathscr{E}; F \models_{\leq \mathscr{R}} G)$, *then* $(\Gamma \vdash \mathscr{E}; \mathsf{split}(F) \models_{\leq \mathscr{R}} \mathsf{split}(G))$.

Proof. Let $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, $(\Gamma; o \vdash_{\mathsf{w}} F, G, \mathsf{split}(F), \mathsf{split}(G) : \mathsf{supp})$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, and $(\Gamma \vdash \mathscr{E}; F \models_{\leq G})$. Let $\mathscr{R}' \leq \mathscr{R}$ and $(\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models \mathsf{split}(F))$. Let $\mathscr{R}'' = \mathsf{final}(\mathscr{R}')$. Then by Lemma 83:

$$\Gamma \vdash \mathscr{E}; \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)); s \models \mathsf{split}(F) \qquad \mathscr{R} \geq \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F))$$

From $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$, we get $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models \mathsf{split}(F) * \mathsf{split}(F))$, by soundness of splitting. Then, by Lemma 83 and because $\mathsf{final}(\mathscr{R}) = \mathsf{final}(\mathscr{R}') = \mathscr{R}''$:

$$\Gamma \vdash \mathscr{E}; \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) * \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)); s \models \mathsf{split}(F) * \mathsf{split}(F)$$

By soundness of merging (Lemma 82) and the assumption $\Gamma \vdash \mathscr{E}; F \models_{\leq \mathscr{R}} G$, we obtain:

$$\Gamma \vdash \mathscr{E}; \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) * \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)); s \models G$$

Then by soundness of splitting:

$$\Gamma \vdash \mathscr{E}; \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) * \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)); s \models \mathsf{split}(G) * \mathsf{split}(G)$$

But then by the minimality property of the support:

$$\begin{array}{lcl} \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) * \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) & \geq & \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(G) * \mathsf{split}(G)) \\ & = & \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(G)) * \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(G)) \end{array}$$

Multiply both sides with $\frac{1}{2}$ (Lemma 24): $\chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) \geq \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(G))$. On the other hand, we have:

$$\Gamma \vdash \mathscr{E}; \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(G)); s \models \mathsf{split}(G)$$

Because $\mathscr{R}' \geq \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(F)) \geq \chi_{\Gamma,\mathscr{E},\mathscr{R}'',s}(\mathsf{split}(G))$, it follows that $\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models \mathsf{split}(G)$. □

T Linear Combinations

The verification rule for calling `join` uses scalar multiplication $fr \cdot F$ of a linear combination fr and a formula F. Linear combinations represent numbers of the forms 1 or $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$. For convenience, we repeat the definitions from Section 6.2.

$$bit \in \{0,1\} \qquad bits ::= 1 \mid bit, bits \qquad fr \in \mathsf{BinFrac} ::= \mathsf{all} \mid \mathsf{fr}() \mid \mathsf{fr}(bits)$$

The scalar multiplication $fr \cdot F$ is defined as follows: $\mathsf{all} \cdot F = F$, $\mathsf{fr}() \cdot F = \mathtt{true}$, $\mathsf{fr}(1) \cdot F = \mathsf{split}(F)$, $\mathsf{fr}(0, bits) \cdot F = \mathsf{fr}(bits) \cdot \mathsf{split}(F)$ and $\mathsf{fr}(1, bits) \cdot F = \mathsf{split}(F) * \mathsf{fr}(bits) \cdot \mathsf{split}(F)$. For instance, $\mathsf{fr}(1,0,1) \cdot F \mathrel{*\!\!-\!\!*} (\mathsf{split}(F) * \mathsf{split}^3(F))$.

Lemma 85 *If* $(\Gamma; v \vdash F : \mathsf{supp})$, *then* $(\Gamma; v; F \vdash fr \cdot F)$.

Proof. By induction on the length of fr, using the split-direction of the split/merge axiom. □

Lemma 86 *If* $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$, $(\Gamma; o \vdash F, G : \mathsf{supp})$, $(\Gamma \vdash \mathscr{E}; \mathscr{R}; s \models F)$ *and* $(\Gamma \vdash \mathscr{E}; F \models_{\leq \mathscr{R}} G)$, *then* $(\Gamma \vdash \mathscr{E}; fr \cdot F \models_{\leq \mathscr{R}} fr \cdot G)$.

Proof. By induction on the length of fr using a similar property for split (Lemma 84), and the fact that splitting preserves supportedness (Lemma 2). □

In order to prove the soundness of our verification rule for joining threads, we need a few more lemmas about scalar multiplication. In particular, we need the distributivity law $\Gamma; v; \mathtt{true} \vdash (fr_1 + fr_2) \cdot F \mathrel{*\!\!-\!\!*} (fr_1 \cdot F) * (fr_2 \cdot F)$.

In order to define the addition on the left-hand side of this law, we mimic expansion to a common denominator as known from addition of rational numbers. The map $[\![\cdot]\!] : \mathsf{BinFrac} \to \mathbb{Q}$ interprets symbolic binary fractions as concrete rationals:

$$[\![\mathsf{all}]\!] \triangleq 1 \quad [\![\mathsf{fr}()]\!] \triangleq 0 \quad [\![\mathsf{fr}(1)]\!] \triangleq \tfrac{1}{2} \quad [\![\mathsf{fr}(0, bits)]\!] \triangleq \tfrac{1}{2}[\![\mathsf{fr}(bits)]\!] \quad [\![\mathsf{fr}(1, bits)]\!] \triangleq \tfrac{1}{2} + \tfrac{1}{2}[\![\mathsf{fr}(bits)]\!]$$

Lemma 87 (Addition of Symbolic Fractions) *If $[\![fr_1]\!] + [\![fr_2]\!] \leq 1$, then there exists a unique symbolic fraction $fr_1 + fr_2$ such that $[\![fr_1 + fr_2]\!] = [\![fr_1]\!] + [\![fr_2]\!]$.*

Proof. If $[\![fr_1]\!] = 0$, we define $fr_1 + fr_2 \triangleq fr_2$. Similarly, in case $[\![fr_2]\!] = 0$, we define $fr_1 + fr_2 \triangleq fr_1$. Let's assume that $[\![fr_1]\!] > 0$ and $[\![fr_2]\!] > 0$. Then also $[\![fr_1]\!] < 1$ and $[\![fr_2]\!] < 1$, because $[\![fr_1]\!] + [\![fr_2]\!] \leq 1$, by assumption. Then, by definition of $[\![\cdot]\!]$, $[\![fr_1]\!] = \sum_{i=1}^{n} bit_{1,i} \cdot \frac{1}{2^i}$ and $[\![fr_2]\!] = \sum_{i=1}^{k} bit_{2,i} \cdot \frac{1}{2^i}$, where $bit_{1,n} = 1$ and $bit_{2,k} = 1$. Suppose that $n \geq k$ (the other case is symmetric). Using standard arithmetic, we can represent $[\![fr_1]\!] + [\![fr_2]\!]$ as $\frac{c}{2^n}$ with $c \leq 2^n$. If $c = 2^n$, we define $fr_1 + fr_2 \triangleq \mathsf{all}$. Otherwise, we use standard arithmetic to rewrite $\frac{c}{2^n}$ to $\sum_{i=1}^{m} bit_i \cdot \frac{1}{2^i}$, where $bit_m = 1$. We then define $fr_1 + fr_2 \triangleq \mathsf{fr}(bit_1, \ldots, bit_m)$. The uniqueness follows from the fact that the above sum representations of concrete binary fractions are unique (by standard arithmetic). □

Lemma 88 (Subtraction of Symbolic Fractions) *If $[\![fr_2]\!] \leq [\![fr_1]\!]$, then there exists a unique symbolic fraction $fr_1 - fr_2$ such that $(fr_1 - fr_2) + fr_2 = fr_1$.*

Proof. If $[\![fr_1]\!] - [\![fr_2]\!] = 0$, we define $fr_1 - fr_2 \triangleq \mathsf{fr}()$. If $[\![fr_1]\!] - [\![fr_2]\!] = 1$, we define $fr_1 - fr_2 \triangleq \mathsf{all}$. Otherwise, we represent $[\![fr_1]\!] - [\![fr_2]\!]$ as $\sum_{i=1}^{n} bit_i \cdot \frac{1}{2^i}$ where $bit_n = 1$, and we define $fr_1 - fr_2 \triangleq \mathsf{fr}(bit_1, \ldots, bit_n)$. By construction, we have $[\![fr_1 - fr_2]\!] + [\![fr_2]\!] = [\![fr_2]\!]$. Then $(fr_1 - fr_2) + fr_2 = fr_2$ because, by our definition of addition, $(fr_1 - fr_2) + fr_2$ is the only symbolic fraction fr such that $[\![fr_1 - fr_2]\!] + [\![fr_2]\!] = [\![fr]\!]$. □

Computing the sum $[\![fr_1]\!] + [\![fr_2]\!]$ is done by first reducing the summands of $[\![fr_1]\!]$ and $[\![fr_2]\!]$ to a common denominator. We will mimic this algorithm on formulas. Part (b) of Lemma 90 below says that the operation that mimics the reduction to a common denominator is an equivalence transformation.

We define an operation $\mathsf{copy}(n, F)$ that $*$-conjoins n copies of formula F:

$$\mathsf{copy}(0, F) \triangleq \mathtt{true} \qquad \mathsf{copy}(n+1, F) \triangleq F * \mathsf{copy}(n, F)$$

Lemma 89 *Let $(\Gamma; v \vdash F : \mathsf{supp})$ and $n \geq 0$.*

(a) $\Gamma; v; \mathtt{true} \vdash \mathsf{copy}(2n, \mathsf{split}(F)) \mathrel{*\!\!-\!\!*} \mathsf{copy}(n, F)$
(b) $\Gamma; v; \mathtt{true} \vdash \mathsf{copy}(2^n, \mathsf{split}^n(F)) \mathrel{*\!\!-\!\!*} F$

Proof. Part (a) uses the split/merge axiom n times. Part (b) by induction on n. For $n = 0$, we have $\mathsf{copy}(2^0, F) = F$. For $n > 0$, we use the induction hypothesis to obtain:

$$\begin{array}{lll} \mathsf{copy}(2^n, \mathsf{split}^n(F)) & \mathrel{*\!\!-\!\!*} & \mathsf{copy}(2, \mathsf{copy}(2^{n-1}, \mathsf{split}^{n-1}(\mathsf{split}(F)))) \\ & \mathrel{*\!\!-\!\!*} & \mathsf{copy}(2, \mathsf{split}(F)) \mathrel{*\!\!-\!\!*} F \end{array}$$

□

Lemma 90 (Reduction to Common Denominator) *Let* $(\Gamma; v \vdash F : \mathsf{supp})$, $|bits| = n \leq m$ *and* $fr = \mathsf{fr}(bits)$.

(a) $[\![fr]\!] = (\sum_{i=1}^{n} bit_i \cdot 2^{m-i}) \cdot \frac{1}{2^m}$
(b) $\Gamma; v; \mathtt{true} \vdash fr \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(\sum_{i=1}^{n} bit_i \cdot 2^{m-i}, \mathsf{split}^m(F))$
(c) $\Gamma; v; \mathtt{true} \vdash fr \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(2^m [\![fr]\!], \mathsf{split}^m(F))$

Proof. Part (c) follow from parts (a) and (b). Parts (a) and (b) are both shown by induction on the structure of *bits*. We do the proof of part (b) in detail: For $bits = 1$, we have $\mathsf{fr}(1) \cdot F = \mathsf{split}(F)$ and on the other hand:

$$\mathsf{copy}(\textstyle\sum_{i=1}^{1} 2^{m-i}, \mathsf{split}^m(F)) = \mathsf{copy}(2^{m-1}, \mathsf{split}^{m-1}(\mathsf{split}(F))) \mathrel{*\!\!-\!\!*} \mathsf{split}(F)$$

For the last equivalence in this chain, we used Lemma 89(b). Suppose now that $bits = (bit_1, bits')$. We derive the following:

$$\begin{array}{ll} & \mathsf{copy}(\sum_{i=2}^{n} bit_i \cdot 2^{m-i}, \mathsf{split}^m(F)) \\ = & \mathsf{copy}(\sum_{i=1}^{n-1} bit_{i+1} \cdot 2^{m-i-1}, \mathsf{split}^m(F)) \\ = & \mathsf{copy}(\frac{1}{2}\sum_{i=1}^{n-1} bit_{i+1} \cdot 2^{m-i}, \mathsf{split}^m(F)) \\ \mathrel{*\!\!-\!\!*} & \mathsf{copy}(\sum_{i=1}^{n-1} bit_{i+1} \cdot 2^{m-i}, \mathsf{split}^{m+1}(F)) \\ \mathrel{*\!\!-\!\!*} & \mathsf{fr}(bits') \cdot \mathsf{split}(F) \end{array}$$

The last step uses the induction hypothesis and the step before Lemma 89(a). Using this equivalence, we now get:

$$\begin{array}{ll} & \mathsf{copy}(\sum_{i=1}^{n} bit_i \cdot 2^{m-i}, \mathsf{split}^m(F)) \\ = & \mathsf{copy}(bit_1 \cdot 2^{m-1}, \mathsf{split}^m(F)) * \mathsf{copy}(\sum_{i=2}^{n} bit_i \cdot 2^{m-i}, \mathsf{split}^m(F)) \\ \mathrel{*\!\!-\!\!*} & \mathsf{copy}(bit_1 \cdot 2^{m-1}, \mathsf{split}^m(F)) * \mathsf{fr}(bits') \cdot \mathsf{split}(F) \end{array}$$

In case $bit_1 = 0$, we have:

$$\begin{array}{ll} & \mathsf{copy}(bit_1 \cdot 2^{m-1}, \mathsf{split}^m(F)) * \mathsf{fr}(bits') \cdot \mathsf{split}(F) \\ = & \mathtt{true} * \mathsf{fr}(bits') \cdot \mathsf{split}(F) \\ \mathrel{*\!\!-\!\!*} & \mathsf{fr}(bits') \cdot \mathsf{split}(F) = \mathsf{fr}(bits) \cdot F \end{array}$$

In case $bit_1 = 1$, we have:

$$\begin{array}{ll} & \mathsf{copy}(bit_1 \cdot 2^{m-1}, \mathsf{split}^m(F)) * \mathsf{fr}(bits') \cdot \mathsf{split}(F) \\ = & \mathsf{copy}(2^{m-1}, \mathsf{split}^m(F)) * \mathsf{fr}(bits') \cdot \mathsf{split}(F) \\ \mathrel{*\!\!-\!\!*} & \mathsf{split}(F) * \mathsf{fr}(bits') \cdot \mathsf{split}(F) = \mathsf{fr}(bits) \cdot F \end{array}$$

For the equivalence on the last line, we used Lemma 89(b). □

Lemma 91 (Monotonicity of Scalar Multiplication) *If* $[\![fr]\!] \geq [\![fr']\!]$ *and* $(\Gamma; v \vdash F : \mathsf{supp})$, *then* $(\Gamma; v; fr \cdot F \vdash fr' \cdot F)$.

Proof. If $[\![fr]\!] = 1$, we know by Lemma 85. If $[\![fr']\!] = 0$, then this is trivial because $fr' \cdot F = \mathtt{true}$. So let's assume that $1 > [\![fr]\!] \geq [\![fr']\!] > 0$. Then $fr = \mathsf{fr}(bits)$ and $fr' = \mathsf{fr}(bits')$ for some $bits, bits'$. Let $m = \max(|bits|, |bits'|)$. By Lemma 90, we obtain $(fr \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(2^m [\![fr]\!], \mathsf{split}^m(F)))$ and $(fr' \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(2^m [\![fr']\!], \mathsf{split}^m(F)))$. But clearly $\mathsf{copy}(2^m [\![fr']\!], \mathsf{split}^m(F))$ is derivable from $\mathsf{copy}(2^m [\![fr]\!], \mathsf{split}^m(F))$ by dropping $([\![fr]\!] - [\![fr']\!])$ copies of $\mathsf{split}^m(F)$. □

Lemma 92 (Distributivity of Scalar Multiplication) *If* $(\Gamma; v \vdash F : \mathsf{supp})$ *and* $fr_1 + fr_2$ *exists, then* $\Gamma; v; \mathtt{true} \vdash (fr_1 + fr_2) \cdot F \mathrel{*\!\!-\!\!*} fr_1 \cdot F * fr_2 \cdot F$.

Proof. If $fr_1 = \mathsf{fr}()$, then the left-hand-side equals $fr_2 \cdot F$ and the right-hand-side equals $\mathtt{true} * fr_2 \cdot F$. These are equivalent. The case $fr_2 = \mathsf{fr}()$ is symmetric. The case where either $fr_1 = \mathsf{all}$ or $fr_2 = \mathsf{all}$ is covered, because then the other one has to be $\mathsf{fr}()$ (otherwise $fr_1 + fr_2$ would not exist). So let's assume that $fr_1 = \mathsf{fr}(bits)$ and $fr_2 = \mathsf{fr}(bits')$ for some *bits* and $bits'$. Let $m = \mathsf{max}(|bits|, |bits'|)$. By Lemma 90, we then know that:

(1) $\Gamma; v; \mathtt{true} \vdash fr_1 \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(2^m[\![fr_1]\!], \mathsf{split}^m(F))$
(2) $\Gamma; v; \mathtt{true} \vdash fr_2 \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(2^m[\![fr_2]\!], \mathsf{split}^m(F))$
(3) $\Gamma; v; \mathtt{true} \vdash (fr_1 + fr_2) \cdot F \mathrel{*\!\!-\!\!*} \mathsf{copy}(2^m[\![fr_1 + fr_2]\!], \mathsf{split}^m(F))$

(In case $fr_1 + fr_2 = \mathsf{all}$, the last equivalence holds by Lemma 89(b).) By definition of $fr_1 + fr_2$, we have that $[\![fr_1 + fr_2]\!] = [\![fr_1]\!] + [\![fr_2]\!]$. Therefore:

$$
\begin{array}{ll}
 & (fr_1 + fr_2) \cdot F \\
\mathrel{*\!\!-\!\!*} & \mathsf{copy}(2^m[\![fr_1 + fr_2]\!], \mathsf{split}^m(F)) \\
= & \mathsf{copy}(2^m[\![fr_1]\!] + 2^m[\![fr_2]\!], \mathsf{split}^m(F)) \\
\mathrel{*\!\!-\!\!*} & \mathsf{copy}(2^m[\![fr_1]\!], \mathsf{split}^m(F)) * \mathsf{copy}(2^m[\![fr_2]\!], \mathsf{split}^m(F)) \\
\mathrel{*\!\!-\!\!*} & fr_1 \cdot F * fr_2 \cdot F
\end{array}
$$

□

U Preservation

Proof of Theorem 5 (Preservation). *If* $(ct : \diamond)$, $(st : \diamond)$ *and* $st \rightarrow_{ct} st'$, *then* $(st' : \diamond)$.

Proof.

(1) $ct : \diamond$ assumption
(2) $st : \diamond$ assumption
(3) $st \rightarrow st'$ assumption

An inspection of the reduction rules shows that *st* is of the following form:

(4) $st = \langle h, ts \mid o \text{ is } (s \text{ in } c) \rangle$

By inverting the last verification rules in the derivation of $st : \diamond$, we obtain $\mathscr{R}_{ts}$, $\mathscr{R}$, Γ, Γ', F, G, fr and G' such that the following statements hold:

(5) $h = \mathscr{R}_{\mathsf{hp}} * (\mathscr{R}_{ts})_{\mathsf{hp}}$
(6) $\mathsf{dom}(\mathscr{R}_{\mathsf{hp}}) = \mathsf{dom}((\mathscr{R}_{ts})_{\mathsf{hp}})$
(7) $\mathscr{R}_{ts} \vdash ts : \diamond$
(8) $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$
(9) $\Gamma \vdash \sigma : \Gamma'$
(10) $\Gamma, \Gamma' \vdash s : \diamond$
(11) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma] : \checkmark$
(12) $\Gamma, \Gamma'; r \vdash \{F\}c : \mathtt{void}\{G\}$
(13) $\mathsf{cfv}(c) \cap \mathsf{dom}(\Gamma') = \emptyset$
(14) $G = fr \cdot G'[o/\mathtt{this}]$
(15) $\mathsf{post}(h(o)_1, \mathtt{run}) = G'$

(16) $\mathscr{R}_{\mathsf{glo}}(o, \texttt{join}) \leq [\![fr]\!]$
(17) $fr = \mathsf{all}$ or $(\exists \mathscr{R}' \geq \mathscr{R})(\Gamma \vdash \mathscr{E}; \mathscr{R}'; s \models G'[o/\texttt{this}])$
(18) $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$ and $\mathsf{dom}(\Gamma') \subseteq \mathsf{LogVar}$

Statement (18) is implied by (11) and (9). All other statements are taken from the premises of (State), (Cons Pool) and (Thread). We now distinguish cases by the reduction rules that can possibly be the reason for $st \to st'$:

Case 1, (Red Dcl):

$$\frac{\ell \notin \mathsf{dom}(s) \quad s' = s[\ell \mapsto \mathsf{df}(T)]}{\langle h, ts \mid o \,\mathsf{is}\, (s \,\mathsf{in}\, T\ \ell; c') \rangle \to \langle h, ts \mid o \,\mathsf{is}\, (s' \,\mathsf{in}\, c') \rangle}$$

In this case, we can further instantiate c and st' as follows:

(1.1) $c = T\ \ell; c'$
(1.2) $st' = \langle h, ts \mid o \,\mathsf{is}\, (s' \,\mathsf{in}\, c') \rangle$
(1.3) $s' \triangleq s[\ell \mapsto \mathsf{df}(T)]$

By bound variable renaming, we may assume:

(1.4) $\ell \notin \mathsf{dom}(\Gamma)$

We define:

(1.5) $\Gamma_\ell \triangleq \Gamma, \ell : T$

We apply weakening (Lemma 39) to judgment (10) and obtain $(\Gamma_\ell, \Gamma' \vdash s : \diamond)$. Moreover, we have $(\Gamma_\ell, \Gamma' \vdash \mathsf{df}(T) : T)$. Using (Stack), we get:

(1.6) $\Gamma_\ell, \Gamma' \vdash s' : \diamond$

By inverting judgment (12), we obtain:

(1.7) $\Gamma_\ell, \Gamma'; r \vdash \{F * \ell == \mathsf{df}(T)\} c' : \texttt{void}\{G\}$
(1.8) $\ell \notin F, G$

Because $\ell \notin \mathsf{fv}(F[\sigma])$, we can apply Lemma 72(c) to (11) in order to extend s to s':

(1.9) $\Gamma_\ell[\sigma] \vdash \mathscr{E}; \mathscr{R}; s' \models F[\sigma] : \checkmark$

We have $[\![\ell]\!]^{h'}_{s'} = s'(\ell) = \mathsf{df}(T) = [\![\mathsf{df}(T)]\!]^{h'}_{s'}$ for any h'. The last of these equalities holds because $\mathsf{df}(T)$ is a closed value. We thus have:

(1.10) $\Gamma_\ell[\sigma] \vdash \mathscr{E}; \mathsf{final}(\mathscr{R}); s' \models \ell == \mathsf{df}(T) : \checkmark$

Because $\mathscr{R} * \mathsf{final}(\mathscr{R}) = \mathscr{R}$ and because (1.9) and (1.10) hold, we obtain:

(1.11) $\Gamma_\ell[\sigma] \vdash \mathscr{E}; \mathscr{R}; s' \models F[\sigma] * \ell == \mathsf{df}(T) : \checkmark$

Now, we apply (Thread) to (1.6), (1.11) and (1.7). We obtain:

(1.12) $\mathscr{R} \vdash o \,\mathsf{is}\, (s' \,\mathsf{in}\, c') : \diamond$

Finally, we apply (Cons Pool) and (State) to (1.12), (7), (6) and (5). We obtain:

(1.13) $st' : \diamond$

Case 2, (Red Fin Dcl):

$$\frac{s(\ell) = v \quad c'' = c'[v/\imath]}{\langle h, ts \mid o \text{ is } (s \text{ in } \texttt{final } T\ \imath\texttt{=}\ell; c')\rangle \rightarrow \langle h, ts \mid o \text{ is } (s \text{ in } c'')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(2.1) $c = \texttt{final } T\ \imath\texttt{=}\ell; c'$
(2.2) $st' = \langle h, ts \mid o \text{ is } (s \text{ in } c'')\rangle$

The last rule in (12)'s derivation is (Fin Dcl). By the premises of this rule, we get:

(2.3) $\Gamma, \Gamma', \imath : T; r \vdash \{F * \imath \texttt{ == } \ell\} c' : \texttt{void}\{G\}$
(2.4) $\Gamma, \Gamma' \vdash \ell : T$
(2.5) $\imath \notin F, G$.

Because $(\Gamma, \Gamma' \vdash s : \diamond)$ and $s(\ell) = v$, we have $(\Gamma, \Gamma' \vdash v : T)$. By substitutivity (Lemma 61), we can apply the substitution $[v/\imath]$ to (2.3), obtaining:

(2.6) $\Gamma, \Gamma'; r \vdash \{F * v \texttt{ == } \ell\} c'' : \texttt{void}\{G\}$

We have $[\![\ell]\!]^{h'}_s = s(\ell) = v = [\![v]\!]^{h'}_{s'}$ for any h'. The last of these equalities holds because v is a closed value (this follows from (10) and (18)). We thus have:

(2.7) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{final}(\mathscr{R}); s \models v \texttt{ == } \ell : \checkmark$

Because $\mathscr{R} * \mathsf{final}(\mathscr{R}) = \mathscr{R}$, we can combine (11) and (2.7) to obtain:

(2.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma] \texttt{ * } v \texttt{ == } \ell : \checkmark$

We apply (Thread) to (2.8) and (2.6), and obtain $\mathscr{R} \vdash o \text{ is } (s \text{ in } c'') : \diamond$. Applying (Cons Pool) and (State) to the previous judgment and (7), (6), (5), we obtain $st' : \diamond$.

Case 3, (Red Unpack):

$$\frac{}{\langle h, ts \mid o \text{ is } (s \text{ in } \texttt{unpack (ex } T\ \alpha\texttt{) (}H\texttt{); } c')\rangle \rightarrow \langle h, ts \mid o \text{ is } (s \text{ in } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(3.1) $c = \texttt{unpack (ex } T\ \alpha\texttt{) (}H\texttt{); } c'$
(3.2) $st' = \langle h, ts \mid o \text{ is } (s \text{ in } c')\rangle$

Define $\Gamma'_\alpha = (\Gamma', \alpha : T)$. The last rule in (12)'s derivation is (Unpack). By the premises of this rule, there exists F' such that:

(3.3) $F = F' * \texttt{(ex } T\ \alpha\texttt{) (}H\texttt{)}$
(3.4) $\Gamma, \Gamma'_\alpha; r \vdash \{F' * H\} c' : \texttt{void}\{G\}$
(3.5) $\alpha \notin F', G$

From (11), we obtain $\mathscr{R}_1, \mathscr{R}_2, \pi$ such that $\mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2$, $(\Gamma[\sigma] \vdash \pi : T[\sigma])$ and:

(3.6) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}_1; s \models F'[\sigma] : \checkmark$
(3.7) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}_2; s \models H[\sigma][\pi/\alpha] : \checkmark$

Define $\sigma_\alpha = (\sigma, \alpha \mapsto \pi)$. Then:

(3.8) $\Gamma \vdash \sigma_\alpha : \Gamma'_\alpha$

Because $\alpha \notin \mathsf{fv}(\Gamma, F', \mathsf{ran}(\sigma))$, statements (3.6) and (3.7) are equivalent to:

(3.9) $\Gamma[\sigma_\alpha] \vdash \mathscr{E};\mathscr{R}_1;s \models F'[\sigma_\alpha] : \checkmark$
(3.10) $\Gamma[\sigma_\alpha] \vdash \mathscr{E};\mathscr{R}_2;s \models H[\sigma_\alpha] : \checkmark$

Thus, we have:

(3.11) $\Gamma[\sigma_\alpha] \vdash \mathscr{E};\mathscr{R};s \models (F' * H)[\sigma_\alpha] : \checkmark$

We can now apply (Thread) to (3.8), (3.11) and (3.4), obtaining $(\mathscr{R} \vdash o \text{ is } (s \text{ in } c') : \diamond)$.

Case 4, (Red Var Set):

$$\frac{s' = s[\ell \mapsto v]}{\langle h, ts \mid o \text{ is } (s \text{ in } \ell = v; c') \rangle \rightarrow \langle h, ts \mid o \text{ is } (s' \text{ in } c') \rangle}$$

In this case, we can further instantiate c and st' as follows:

(4.1) $c = \ell = v; c'$
(4.2) $st' = \langle h, ts \mid o \text{ is } (s' \text{ in } c') \rangle$
(4.3) $s' \stackrel{\Delta}{=} s[\ell \mapsto v]$

The last rules in (12)'s derivation are (Seq) preceded by (Var Set). From the premises of these rules, we obtain an F' such that:

(4.4) $\Gamma, \Gamma' \vdash v : (\Gamma, \Gamma')(\ell)$
(4.5) $\Gamma, \Gamma'; r \vdash \{F'\}\ell = v\{F' * \ell == v\}$
(4.6) $\Gamma, \Gamma'; r; F \vdash F'$
(4.7) $\Gamma, \Gamma'; r \vdash \{F' * \ell == v\}c' : \texttt{void}\{G\}$
(4.8) $\ell \notin F'$

Because $(\Gamma, \Gamma' \vdash s : \diamond)$ and $(\Gamma, \Gamma' \vdash v : (\Gamma, \Gamma')(\ell))$, we have:

(4.9) $\Gamma, \Gamma' \vdash s' : \diamond$

Applying substitutivity (Lemma 49) to (4.6), we obtain $(\Gamma[\sigma]; r; F[\sigma] \vdash F'[\sigma])$. By soundness of logical consequence (Theorem 4), we can compose $(\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s \models F[\sigma] : \checkmark)$ and $(\Gamma[\sigma]; r; F[\sigma] \vdash F'[\sigma])$ to obtain $(\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s \models F'[\sigma] : \checkmark)$. Because $\ell \notin F'$, we can modify s at argument ℓ (part (c) of Lemma 73) without destroying semantic validity. We obtain:

(4.10) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s' \models F'[\sigma] : \checkmark$

We have $[\![\ell]\!]^{h'}_{s'} = s'(\ell) = v = [\![v]\!]^{h'}_{s'}$ for any heap h'. The last of these equalities holds because $\mathsf{dom}(\Gamma, \Gamma') \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar} \cup \mathsf{LogVar}$ and thus v is not a variable but a closed value. From $[\![\ell]\!]^{h'}_{s'} = [\![v]\!]^{h'}_{s'}$ it follows that $(\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{final}(\mathscr{R}); s' \models \ell == v)$. In combination with (4.10) and $\mathscr{R} * \mathsf{final}(\mathscr{R}) = \mathscr{R}$, we then obtain:

(4.11) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s' \models F'[\sigma] * \ell == v : \checkmark$

We apply (Thread) to (4.9), (4.11) and (4.7) to obtain $(\mathscr{R} \vdash o \text{ is } (s' \text{ in } c') : \diamond)$. Then we apply (Cons Pool) and (State) to $(\mathscr{R} \vdash o \text{ is } (s' \text{ in } c') : \diamond)$, (7), (6), (5). We obtain $st' : \diamond$.

Case 5, (Red Op):

$$\frac{\mathsf{arity}(op) = |\overline{v}| \quad [\![op]\!]^h(\overline{v}) = w \quad s' = s[\ell \mapsto w]}{\langle h, ts \mid o \text{ is } (s \text{ in } \ell = op(\overline{v}); c')\rangle \rightarrow \langle h, ts \mid o \text{ is } (s' \text{ in } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(5.1) $c = \ell = op(\overline{v}); c'$
(5.2) $st' = \langle h, ts \mid o \text{ is } (s' \text{ in } c')\rangle$

The last rules in (12)'s derivation are (Seq) preceded by (Op). From the premises of these rules, we obtain an F' such that:

(5.3) $\Gamma, \Gamma' \vdash op(\overline{v}) : (\Gamma, \Gamma')(\ell)$
(5.4) $\Gamma, \Gamma'; r; F \vdash F'$
(5.5) $\Gamma, \Gamma'; r \vdash \{F' * \ell == op(\overline{v})\}c' : \mathtt{void}\{G\}$
(5.6) $\ell \notin F'$

Let $h' = \{ (p, (\Gamma(p), \emptyset)) \mid p \in \mathsf{dom}(h) \}$. Then $(\Gamma, \Gamma')_{\mathsf{hp}} \vdash h' : \diamond$. On the other hand, we have $\Gamma_{\mathsf{hp}}[\sigma] \vdash h : \diamond$. Thus, $\mathsf{fst} \circ h = (\mathsf{fst} \circ h')[\sigma]$. By axioms (b) and (c) for operator semantics, it follows that $w = [\![op]\!]^h(\overline{v}) = [\![op]\!]^{h'}(\overline{v}) = [\![op(\overline{v})]\!]^{h'}_s$. We can therefore apply Lemma 67 ("expression semantics preserves typing") to (5.3) and obtain $(\Gamma, \Gamma' \vdash w : (\Gamma, \Gamma')(\ell))$. Thus:

(5.7) $\Gamma, \Gamma' \vdash s' : \diamond$

Applying substitutivity (Lemma 49) to (5.4), we obtain $(\Gamma[\sigma]; r; F[\sigma] \vdash F'[\sigma])$. By soundness of logical consequence (Theorem 4), we can compose $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F[\sigma] : \checkmark)$ and $(\Gamma[\sigma]; r; F[\sigma] \vdash F'[\sigma])$ to obtain $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] : \checkmark)$. Because $\ell \notin F'$, we can modify s at argument ℓ (part (c) of Lemma 73) without destroying semantic validity. We obtain:

(5.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s' \models F'[\sigma] : \checkmark$

We have $[\![\ell]\!]^{h'}_{s'} = s'(\ell) = w = [\![op(\overline{v})]\!]^{h'}_{s'}$. By axiom (b) for operator semantics, these equations still hold if we replace h' by $\mathsf{final}(\mathscr{R})_{\mathsf{hp}}$. Therefore, $(\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{final}(\mathscr{R}); s' \models \ell == op(\overline{v}))$. In combination with (5.8) and $\mathscr{R} * \mathsf{final}(\mathscr{R}) = \mathscr{R}$, we then obtain:

(5.9) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s' \models F'[\sigma] * \ell == op(\overline{v}) : \checkmark$

We apply (Thread) to (5.7), (5.9) and (5.5) to obtain $(\mathscr{R} \vdash o \text{ is } (s' \text{ in } c') : \diamond)$. Then we apply (Cons Pool) and (State) to $(\mathscr{R} \vdash o \text{ is } (s' \text{ in } c') : \diamond)$, (7), (6), (5). We obtain $st' : \diamond$.

Case 6, (Red Get):

$$\frac{s' = s[\ell \mapsto h(p)_2(f)]}{\langle h, ts \mid o \text{ is } (s \text{ in } \ell = p.f; c')\rangle \rightarrow \langle h, ts \mid o \text{ is } (s' \text{ in } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(6.1) $c = \ell = p.f; c'$
(6.2) $st' = \langle h, ts \mid o \text{ is } (s' \text{ in } c')\rangle$
(6.3) $s' = s[\ell \mapsto h(p)_2(f)]$

The last rules in (12)'s derivation are (Seq) preceded by (Get). From the premises of these rules, we obtain F', E and E' such that the following statements hold:

(6.4) $\Gamma,\Gamma';r;F \vdash F'$
(6.5) $\Gamma,\Gamma';r;F' \vdash E$
(6.6) $\Gamma,\Gamma' \vdash p.f : (\Gamma)(\ell)$
(6.7) $\ell \notin F'$
(6.8) $\Gamma,\Gamma';r \vdash \{F'\}\ell = p.f\{F' * E'\}$
(6.9) $(E,E') = (\texttt{PointsTo(}p\texttt{[}f\texttt{]},\pi,u\texttt{)},\ \ell == u)$ or $(E,E') = (\texttt{Pure(}p.f\texttt{)},\ \ell == p.f)$
(6.10) $\Gamma,\Gamma';r \vdash \{F' * E'\}c' : \texttt{void}\{G\}$

Let $h' = \mathscr{R}_{\mathsf{hp}}$. From (6.4), (6.5) and (6.9), we deduce that $h'(p)_2(f)$ is defined. Because $h' \leq h$, it must be the case that $h'(p)_2(f) = h(p)_2(f)$. From (6.6), we obtain that $(\Gamma[\sigma] \vdash p.f : \Gamma(\ell)[\sigma])$. Because $(\Gamma[\sigma] \vdash h' : \diamond)$, we then get $(\Gamma[\sigma] \vdash h'(p)_2(f) : \Gamma(\ell)[\sigma])$. Because $h'(p)_2(f) = h(p)_2(f)$, it follows that $(\Gamma[\sigma] \vdash h(p)_2(f) : \Gamma(\ell)[\sigma])$. Then $(\Gamma,\Gamma' \vdash h(p)_2(f) : \Gamma(\ell))$, by Lemma 43. Thus, we have:

(6.11) $\Gamma,\Gamma' \vdash s' : \diamond$

Like in the previous proof cases, soundness of logical consequence (Theorem 4) yields:

(6.12) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s' \models F'[\sigma] : \checkmark$

Similarly, we obtain:

(6.13) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s' \models E[\sigma] : \checkmark$

To complete this proof case, it suffices to show the following:

(6.14) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s' \models F'[\sigma] * E'[\sigma] : \checkmark$ goal

We split cases according to (6.9).

Case 6.1, $(E,E') = (\texttt{PointsTo(}p\texttt{[}f\texttt{]},\pi,u\texttt{)},\ \ell == u)$: From (6.13), we get $h'(p)_2(f) = [\![u]\!]^{h'}_{s'}$. On the other hand, we have $[\![\ell]\!]^{h'}_{s'} = s'(\ell) = h(p)_2(f)$. Therefore, $[\![\ell]\!]^{h'}_{s'} = h(p)_2(f) = h'(p)_2(f) = [\![u]\!]^{h'}_{s'}$. It follows that:

(6.1.1) $\Gamma[\sigma] \vdash \mathscr{E};\mathsf{final}(\mathscr{R});s' \models \ell == u : \checkmark$

Because $\mathscr{R} * \mathsf{final}(\mathscr{R}) = \mathscr{R}$, we can combine (6.12) and (6.1.1) to obtain:

(6.1.2) $\Gamma,\Gamma' \vdash \mathscr{E};\mathscr{R};s' \models F'[\sigma] * \ell == u : \checkmark$

Because $E'[\sigma] = \ell == u$, we have established our goal (6.14).

Case 6.2, $(E,E') = (\texttt{Pure(}p.f\texttt{)},\ \ell == p.f)$: Let $\mathscr{Q} = \mathscr{R}_{\mathsf{glo}}$. From (6.13), we know that $\mathscr{Q}(p,f) < 1$. We have $[\![\ell]\!]^{h'}_{s'} = s'(\ell) = h(p)_2(f) = h'(p)_2(f) = [\![p.f]\!]^{h'}_{s'}$. From this equation and $\mathscr{Q}(p,f) < 1$, we obtain that $(\Gamma[\sigma] \vdash \mathscr{E};\mathsf{final}(\mathscr{R});s' \models \ell == p.f : \checkmark)$. In combination with (6.12), we then get:

(6.2.1) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s' \models F'[\sigma] * \ell == p.f : \checkmark$

Because $E'[\sigma] = \ell == p.f$ we have established our goal (6.14).

Case 7, (Red Set):

$$\frac{h' = h[p.f \mapsto w]}{\langle h, ts \mid o \text{ is } (s \text{ in } \mathit{fin}\ p.f = w; c')\rangle \rightarrow \langle h', ts \mid o \text{ is } (s \text{ in } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(7.1) $c = \mathit{fin}\ p.f = w; c'$
(7.2) $st' = \langle h', ts \mid o \text{ is } (s \text{ in } c')\rangle$
(7.3) $h' = h[p.f \mapsto w]$

The last rules in (12)'s derivation are (Seq) preceded by (Fld Set). From the rule premises, we obtain F' and E such that:

(7.4) $\Gamma, \Gamma'; r; F \vdash F' * \texttt{PointsTo}(p\texttt{[}f\texttt{]}, 1, T)$
(7.5) $\Gamma, \Gamma' \vdash p : C\texttt{<}\bar{\pi}\texttt{>}$
(7.6) $T\ f \in \mathsf{fld}(C\texttt{<}\bar{\pi}\texttt{>})$
(7.7) $\Gamma, \Gamma' \vdash w : T$
(7.8) $(\mathit{fin}, E) = (\varepsilon, \texttt{PointsTo}(p\texttt{[}f\texttt{]}, 1, w))$ or $(\mathit{fin}, E) = (\texttt{final},\ p.f\ \texttt{==}\ w)$
(7.9) $\Gamma, \Gamma'; r \vdash \{F' * E\} c' : \texttt{void}\{G\}$

Like in the previous proof cases, soundness of logical consequence (Theorem 4) yields:

(7.10) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] * \texttt{PointsTo}(p\texttt{[}f\texttt{]}, 1, T[\sigma]) : \checkmark$

This means there are $\mathscr{R}', \mathscr{R}''$ such that $\mathscr{R} = \mathscr{R}' * \mathscr{R}''$ and:

(7.11) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}'; s \models F'[\sigma] : \checkmark$
(7.12) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}''; s \models \texttt{PointsTo}(p\texttt{[}f\texttt{]}, 1, T[\sigma]) : \checkmark$

By inverting $(\Gamma, \Gamma' \vdash p : C\texttt{<}\bar{\pi}\texttt{>})$ we obtain $(\Gamma, \Gamma')(p) <: C\texttt{<}\bar{\pi}\texttt{>}$. Because p is an object id and $\mathsf{dom}(\Gamma') \subseteq \mathsf{LogVar}$, we know that $p \notin \mathsf{dom}(\Gamma')$. Therefore, $\Gamma(p) <: C\texttt{<}\bar{\pi}\texttt{>}$. Let $h'' = \mathscr{R}''_{\mathsf{hp}}$. Because $(\Gamma[\sigma] \vdash h'' : \diamond)$, we know that $\Gamma[\sigma] = \mathsf{fst} \circ h''$. From $\Gamma[\sigma] = \mathsf{fst} \circ h''$ and $\Gamma(p) <: C\texttt{<}\bar{\pi}\texttt{>}$ it follows that $h''(p)_1 = (\Gamma[\sigma])(p) <: C\texttt{<}\bar{\pi}\texttt{>}[\sigma]$. From this and (7.6) it follows that $T[\sigma]\ f \in \mathsf{fld}(h''(p)_1)$. Applying substitutivity to (7.7), we get $(\Gamma[\sigma] \vdash w : T[\sigma])$. Furthermore, we have $\mathscr{R}''_{\mathsf{loc}}(p, f) = 1$ by (7.12).

We now split cases according to (7.8).

Case 7.1, $E = \texttt{PointsTo}(p\texttt{[}f\texttt{]}, 1, w)$: In this case, we have:

(7.1.1) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}''[p.f \mapsto w]; s \models E[\sigma] : \checkmark$

By Lemma 28, we know that $\mathscr{R}' \# \mathscr{R}''[p.f \mapsto w]$ and $\mathscr{R}' * \mathscr{R}''[p.f \mapsto w] = \mathscr{R}[p.f \mapsto w]$. From (7.11) and (7.1.1), it then follows that:

(7.1.2) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}[p.f \mapsto w]; s \models F'[\sigma] * E[\sigma] : \checkmark$

From (7.1.2) and (7.9), it follows that:

(7.1.3) $\mathscr{R}[p.f \mapsto w] \vdash o \text{ is } (s \text{ in } c') : \diamond$

By Lemma 28, we know that $\mathscr{R}_{ts} \# \mathscr{R}[p.f \mapsto w]$ and $\mathscr{R}_{ts} * \mathscr{R}[p.f \mapsto w] = (\mathscr{R}_{ts} * \mathscr{R})[p.f \mapsto w]$. From (7) and (7.1.3) it follows that:

(7.1.4) $(\mathscr{R}_{ts} * \mathscr{R})[p.f \mapsto w] \vdash ts \mid o \text{ is } (s \text{ in } c') : \diamond$

We have $(\mathscr{R}_{ts} * \mathscr{R})[p.f \mapsto w]_{\mathsf{hp}} = h[p.f \mapsto w] = h'$. Thus, by (State):

(7.1.5) $\langle h', ts \mid o \text{ is } (s \text{ in } c')\rangle : \diamond$

Case 7.2, $E = p.f$ == w: In this case, we have:

(7.2.1) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{finalize}(p.f, w, \mathscr{R}''); s \models E[\sigma] : \checkmark$

By applying Lemma 77 to (7.11), we get:

(7.2.2) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{finalize}(p.f, w, \mathscr{R}'); s \models F'[\sigma] : \checkmark$

Because $\mathsf{finalize}(p.f, w, \mathscr{R}') * \mathsf{finalize}(p.f, w, \mathscr{R}'') = \mathsf{finalize}(p.f, w, \mathscr{R})$, we obtain:

(7.2.3) $\Gamma[\sigma] \vdash \mathscr{E}; \mathsf{finalize}(p.f, w, \mathscr{R}); s \models F'[\sigma] * E[\sigma] : \checkmark$

From (7.2.3) and (7.9), it follows that:

(7.2.4) $\mathsf{finalize}(p.f, w, \mathscr{R}) \vdash o \text{ is } (s \text{ in } c') : \diamond$

Applying Lemma 77 to (7), we get:

(7.2.5) $\mathsf{finalize}(p.f, w, \mathscr{R}_{ts}) \vdash ts : \diamond$

Because $\mathsf{finalize}(p.f, w, \mathscr{R}_{ts}) * \mathsf{finalize}(p.f, w, \mathscr{R}) = \mathsf{finalize}(p.f, w, \mathscr{R}_{ts} * \mathscr{R})$, we get:

(7.2.6) $\mathsf{finalize}(p.f, w, (\mathscr{R}_{ts} * \mathscr{R})) \vdash ts \mid o \text{ is } (s \text{ in } c') : \diamond$

We have $\mathsf{finalize}(p.f, w, (\mathscr{R}_{ts} * \mathscr{R}))_{\mathsf{hp}} = h[p.f \mapsto w] = h'$. Thus, by (State):

(7.2.7) $\langle h', ts \mid o \text{ is } (s \text{ in } c') \rangle : \diamond$

Case 8, (Red Cast):

$$\frac{h(v)_1 <: T \quad s' = s[\ell \mapsto v]}{\langle h, ts \mid o \text{ is } (s \text{ in } \ell = (T)v; c') \rangle \rightarrow \langle h, ts \mid o \text{ is } (s' \text{ in } c') \rangle}$$

In this case, we can further instantiate c and st' as follows:

(8.1) $c = \ell = (T)v; c'$
(8.2) $st' = \langle h, ts \mid o \text{ is } (s' \text{ in } c') \rangle$
(8.3) $h(v)_1 <: T$

The last rules in (12)'s derivation are (Seq) preceded by (Cast). From the rule premises, we obtain F' such that:

(8.4) $\Gamma, \Gamma'; r; F \vdash F'$
(8.5) $T <: \Gamma(\ell) <: \texttt{Object}$
(8.6) $\Gamma, \Gamma' \vdash v : \texttt{Object}$
(8.7) $\ell \notin F'$
(8.8) $\Gamma, \Gamma'; r \vdash \{F' * \ell == v\} c' : \texttt{void}\{G\}$

Because $h(v)_1 <: T$, we have $(\Gamma[\sigma] \vdash v : T)$. Because $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$, it follows that $(\Gamma_{\mathsf{hp}}[\sigma] \vdash v : T)$. Because $\Gamma_{\mathsf{hp}} \vdash \diamond$, by Lemma 1, we know that $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$, thus $\Gamma_{\mathsf{hp}}[\sigma] = \Gamma_{\mathsf{hp}}$, thus $(\Gamma_{\mathsf{hp}} \vdash v : T)$. As a result, $(\Gamma, \Gamma' \vdash s' : T)$. The remainder of this proof case is like the proof case for (Red Var Set).

Case 9, (Red New):

$$\frac{p \notin \mathsf{dom}(h) \quad h' = h[p \mapsto (C\texttt{<}\overline{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\overline{\pi}\texttt{>}))] \quad s' = s[\ell \mapsto p]}{\langle h, ts \mid o \text{ is } (s \text{ in } \ell\texttt{=new } C\texttt{<}\overline{\pi}\texttt{>}; c')\rangle \rightarrow \langle h', ts \mid o \text{ is } (s' \text{ in } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(9.1) $c = \ell\texttt{=new } C\texttt{<}\overline{\pi}\texttt{>}; c'$
(9.2) $st' = \langle h, ts \mid o \text{ is } (s' \text{ in } c')\rangle$
(9.3) $h' = h[p \mapsto (C\texttt{<}\overline{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\overline{\pi}\texttt{>}))]$
(9.4) $s' = s[\ell \mapsto p]$

The last rules in (12)'s derivation are (Seq) preceded by (New). From the premises of these rules, we obtain:

(9.5) $C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} \in ct$
(9.6) $\Gamma, \Gamma' \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}]$
(9.7) $C\texttt{<}\overline{\pi}\texttt{>} <: \Gamma(\ell)$
(9.8) $\Gamma, \Gamma'; r; F \vdash F'$
(9.9) $\Gamma, \Gamma'; r \vdash \{F' * \ell\texttt{.init} * C \texttt{ isclassof } \ell\} c : \texttt{void}\{G\}$
(9.10) $\ell \notin F'$

By substitutivity, we get $(\Gamma[\sigma] \vdash \overline{\pi}[\sigma] : \overline{T}[\overline{\pi}/\overline{\alpha}][\sigma])$. Because $\mathsf{dom}(\Gamma') \cap \mathsf{fv}(\overline{\pi}) = \emptyset$ (assumption (13)) and $\mathsf{dom}(\Gamma') = \mathsf{dom}(\sigma)$, we get $(\Gamma[\sigma] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}])$. Because $\mathsf{dom}(\Gamma) \subseteq \mathsf{ObjId} \cup \mathsf{RdWrVar}$, it follows that $(\Gamma_{\mathsf{hp}}[\sigma] \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}])$. Because $\Gamma_{\mathsf{hp}} \vdash \diamond$ (Lemma 1), we know that $\mathsf{fv}(\Gamma_{\mathsf{hp}}) = \emptyset$, thus $\Gamma_{\mathsf{hp}}[\sigma] = \Gamma_{\mathsf{hp}}$, thus $(\Gamma_{\mathsf{hp}} \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}])$, thus:

(9.11) $\Gamma_{\mathsf{hp}} \vdash C\texttt{<}\overline{\pi}\texttt{>} : \diamond$

Let $\Gamma_p = (\Gamma, p : C\texttt{<}\overline{\pi}\texttt{>})$ and $\mathscr{R}' = \mathsf{initrsc}(\Gamma[\sigma], p, C\texttt{<}\overline{\pi}\texttt{>}, \mathscr{R}_{\mathsf{glo}})$. By resource axiom (d) for $\mathscr{R}$, we know that $\mathscr{R}_{\mathsf{glo}}(p,k) = 1$ for all k in $\mathsf{FieldId} \cup \{\texttt{join}\}$. Thus, the premises for Lemma 79 are satisfied and we obtain:

(9.12) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathscr{R}'; s \models p\texttt{.init} : \checkmark$

Furthermore, we have:

(9.13) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathsf{final}(\mathscr{R}'); s \models C \texttt{ isclassof } p : \checkmark$

It follows that:

(9.14) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathscr{R}'; s \models p\texttt{.init} * C \texttt{ isclassof } p : \checkmark$

Like in the previous proof cases, by soundness of logical consequence, we also have:

(9.15) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] : \checkmark$

By resource monotonicity (Lemma 71), it follows that:

(9.16) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathscr{R} \nearrow \mathscr{R}'; s \models F'[\sigma] : \checkmark$

By resource axiom (d) for $\mathscr{R}$, we have $\mathscr{R}_{\mathsf{loc}}(p,k) = 0$ for all $k \in \mathsf{FieldId} \cup \{\texttt{join}\}$. It follows that $(\mathscr{R} \nearrow \mathscr{R}')\#\mathscr{R}'$. Furthermore, $(\mathscr{R} \nearrow \mathscr{R}') * \mathscr{R}' = \mathscr{R} * \mathscr{R}'$ by Lemma 27(f). Thus:

(9.17) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s \models F'[\sigma] * p\texttt{.init} * C \texttt{ isclassof } p : \checkmark$

Because ℓ does not occur in $F'[\sigma] * p\texttt{.init} * C \texttt{ isclassof } p$, we can update s at ℓ:

(9.18) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s' \models F'[\sigma] * p.\texttt{init} * C\ \texttt{isclassof}\ p : \checkmark$

Because $s'(\ell) = p$, then:

(9.19) $\Gamma_p[\sigma] \vdash \mathscr{E}; \mathscr{R} * \mathscr{R}'; s' \models F'[\sigma] * \ell.\texttt{init} * C\ \texttt{isclassof}\ \ell : \checkmark$

Because (9.11), (9.19) and (9.9) hold, we can apply (Thread) to obtain:

(9.20) $\mathscr{R} * \mathscr{R}' \vdash o\ \mathsf{is}\ (s'\ \mathsf{in}\ c') : \diamond$

Applying resource monotonicity (Lemma 71) to (7), we get:

(9.21) $\mathscr{R}_{ts} \nearrow \mathscr{R}' \vdash ts : \diamond$

By resource axiom (d) for $\mathscr{R}_{ts}$, we know that $(\mathscr{R}_{ts})_{\mathsf{loc}}(p,k) = 0$ for all k in $\mathsf{FieldId} \cup \{\texttt{join}\}$. It follows that $(\mathscr{R}_{ts} \nearrow \mathscr{R}')\#\mathscr{R} * \mathscr{R}'$. By Lemma 27(f), $(\mathscr{R}_{ts} \nearrow \mathscr{R}') * \mathscr{R} * \mathscr{R}' = \mathscr{R}_{ts} * \mathscr{R} * \mathscr{R}'$. We can, thus, apply (Cons Pool) to obtain:

(9.22) $\mathscr{R}_{ts} * \mathscr{R} * \mathscr{R}' \vdash ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c') : \diamond$

We have $(\mathscr{R}_{ts} * \mathscr{R} * \mathscr{R}')_{\mathsf{hp}} = \mathscr{R}'_{\mathsf{hp}} = h[p \mapsto (C\texttt{<}\bar{\pi}\texttt{>}, \mathsf{init}(C\texttt{<}\bar{\pi}\texttt{>}))] = h'$. Thus, we can apply (State) to obtain:

(9.23) $\langle h', ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c') \rangle : \diamond$

Case 10, (Red If True):

$$\langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{if (true)\{}c'\texttt{\}else\{}c''\texttt{\}}; c''')\rangle \rightarrow \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c'; c''')\rangle$$

In this case, we can further instantiate c and st' as follows:

(10.1) $c = \texttt{if (true)\{}c'\texttt{\}else\{}c''\texttt{\}}; c'''$
(10.2) $st' = \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ c'; c''')\rangle$

The last rules in (12)'s derivation are (Seq) preceded by (If). From the premises of these rules, we obtain:

(10.3) $\Gamma, \Gamma'; r; F \vdash F'$
(10.4) $\Gamma, \Gamma'; r \vdash \{F' * \texttt{true}\}c' : \texttt{void}\{E\}$
(10.5) $\Gamma, \Gamma'; r \vdash \{F' * \texttt{!true}\}c'' : \texttt{void}\{E\}$
(10.6) $\Gamma, \Gamma'; r \vdash \{E\}c''' : \texttt{void}\{G\}$

By Lemma 66, we obtain:

(10.7) $\Gamma, \Gamma'; r \vdash \{F'\}c'; c''' : \texttt{void}\{G\}$

By soundness of logical consequence (Theorem 4):

(10.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] * \texttt{true} : \checkmark$

The rest is routine.

Case 11, (Red If False): Similar to proof case (Red If True).

Case 12, (Red Call):

$$\frac{m \notin \{\texttt{fork}, \texttt{join}\} \quad h(p)_1 = C\texttt{<}\overline{\pi'}\texttt{>} \quad \mathsf{mbody}(m, C\texttt{<}\overline{\pi'}\texttt{>}) = \texttt{<}\overline{\alpha}, \overline{\alpha'}\texttt{>}(\iota_0, \overline{\iota}).c_m \quad c'' = c_m[\overline{\pi}/\overline{\alpha}, p/\iota_0, \overline{v}/\overline{\iota}]}{\langle h, ts \mid o \text{ is } (s \text{ in } \ell = p.m\texttt{<}\overline{\pi}\texttt{>}(\overline{v}); c')\rangle \rightarrow \langle h, ts \mid o \text{ is } (s \text{ in } \ell \leftarrow c''; c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(12.1) $c = \ell = p.m\texttt{<}\overline{\pi}\texttt{>}(\overline{v}); c'$
(12.2) $st' = \langle h, ts \mid o \text{ is } (s \text{ in } \ell \leftarrow c''; c')\rangle$

The last rules in (12)'s derivation are (Seq) preceded by (Call). From the rule premises, we obtain:

(12.3) $\Gamma, \Gamma'; r; F \vdash F' * E[\sigma']$
(12.4) $\mathsf{mtype}(m, t\texttt{<}\overline{\pi''}\texttt{>}) = \mathit{fin}\ \texttt{<}\overline{T}\ \overline{\alpha}\texttt{>}\,\texttt{req}\,E; \texttt{ens}\ (\texttt{ex}\,U\,\alpha'')(H); U\ m(t\texttt{<}\overline{\pi''}\texttt{>}\iota_0, \overline{V}\,\overline{\iota})$
(12.5) $\sigma' = (p/\iota_0, \overline{\pi}/\overline{\alpha}, \overline{v}/\overline{\iota})$
(12.6) $\Gamma, \Gamma' \vdash p, \overline{\pi}, \overline{v} : t\texttt{<}\overline{\pi''}\texttt{>}, \overline{T}[\sigma'], \overline{V}[\sigma']$
(12.7) $U[\sigma'] <: \Gamma(\ell)$
(12.8) $\Gamma, \Gamma'; r \vdash \{F' * (\texttt{ex}\,U[\sigma']\ \alpha'')(\alpha'' == \ell * H[\sigma'])\}c' : \texttt{void}\{G\}$
(12.9) $\ell \notin F'$

From (11) and (12.3), we obtain:[6]

(12.10) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] * E[\sigma'; \sigma] : \checkmark$

From $\Gamma_{\mathsf{hp}} \vdash \diamond$ (Lemma 1), it follows that:

(12.11) $\Gamma_{\mathsf{hp}} \vdash C\texttt{<}\overline{\pi'}\texttt{>} : \diamond$

From (12.6), we obtain $(\Gamma[\sigma] \vdash p : t\texttt{<}\overline{\pi''}\texttt{>}[\sigma])$, by substitutivity. Then $\Gamma[\sigma](p) <: t\texttt{<}\overline{\pi''}\texttt{>}[\sigma]$. But $\Gamma[\sigma](p) = h(p)_1 = C\texttt{<}\overline{\pi'}\texttt{>}$. Thus, $C\texttt{<}\overline{\pi'}\texttt{>} <: t\texttt{<}\overline{\pi''}\texttt{>}[\sigma]$. Because $\Gamma_{\mathsf{hp}} \vdash C\texttt{<}\overline{\pi'}\texttt{>} : \diamond$, we know that $\mathsf{fv}(C\texttt{<}\overline{\pi'}\texttt{>}) = \emptyset$, thus $C\texttt{<}\overline{\pi'}\texttt{>}[\sigma] = C\texttt{<}\overline{\pi'}\texttt{>}$, thus $C\texttt{<}\overline{\pi'}\texttt{>}[\sigma] <: t\texttt{<}\overline{\pi''}\texttt{>}[\sigma]$. By Lemma 41(c), it follows that $C\texttt{<}\overline{\pi'}\texttt{>} <: t\texttt{<}\overline{\pi''}\texttt{>}$. Therefore, by monotonicity of mtype (Lemma 58), $\Gamma_{\mathsf{hp}} \vdash \mathsf{mtype}(m, C\texttt{<}\overline{\pi'}\texttt{>}) <: \mathsf{mtype}(m, t\texttt{<}\overline{\pi''}\texttt{>})$. Then, by definition of method subtyping, we get:

(12.12) $\mathsf{mtype}(m, C\texttt{<}\overline{\pi'}\texttt{>}) = \mathit{fin}'\ \texttt{<}\overline{T'}\ \overline{\alpha}, \overline{W}\ \overline{\alpha'}\texttt{>}\,\texttt{req}\,E'; \texttt{ens}\ (\texttt{ex}\,U'\,\alpha'')(H'); U'\ m(C\texttt{<}\overline{\pi'}\texttt{>}\iota_0, \overline{V'}\,\overline{\iota})$
(12.13) $\overline{T} <: \overline{T'}, U' <: U, \overline{V} <: \overline{V'}$
(12.14) $\Gamma_{\mathsf{hp}}, \iota_0 : C\texttt{<}\overline{\pi'}\texttt{>}; \iota_0; \texttt{true} \vdash (\texttt{fa}\,\overline{T}\,\overline{\alpha})(\texttt{fa}\,\overline{V}\,\overline{\iota})(E \texttt{-*} (\texttt{ex}\,\overline{W}\,\overline{\alpha'})(E' * (\texttt{fa}\,U'\,\alpha'')(H' \texttt{-*} H)))$

To abbreviate, let $H'' = (\texttt{fa}\,U'\,\alpha'')(H' \texttt{-*} H)$. Applying substitutivity and (Fa Elim) to (12.14), we obtain:

(12.15) $\Gamma_{\mathsf{hp}}; p; \texttt{true} \vdash E[\sigma'; \sigma] \texttt{-*} (\texttt{ex}\,\overline{W}[\sigma'; \sigma]\,\overline{\alpha'})(E'[\sigma'; \sigma] * H''[\sigma'; \sigma])$

From (12.10) and (12.15), it follows that there exist $\overline{\pi'''}$ and σ'' such that:

(12.16) $\sigma'' = (\sigma, \overline{\alpha'} \mapsto \overline{\pi'''})$
(12.17) $\Gamma[\sigma''] \vdash \overline{\pi'''} : \overline{W}[\sigma'; \sigma'']$
(12.18) $\Gamma[\sigma''] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma''] * E'[\sigma'; \sigma''] * H''[\sigma'; \sigma''] : \checkmark$

[6] We use the semicolon for substitution composition: $(\sigma'; \sigma)(x) \triangleq \sigma'(x)[\sigma]$

Let $\Gamma'' = (\Gamma', \bar{\alpha} : \bar{W}[\sigma'])$. By (12.17), we get:

(12.19) $\Gamma \vdash \sigma'' : \Gamma''$

We have assumed that $ct : \diamond$. We take the premise of rule (Mth) for m in C and substitute actual class parameters for formal class parameters and actual method parameters for formal method parameters:

(12.20) $\Gamma, \Gamma''; p \vdash \{E'[\sigma'] * p \neq \texttt{null}\} c'' : U'[\sigma']\{(\texttt{ex}\, U'[\sigma']\ \alpha'')(H'[\sigma'])\}$

By the frame property (Lemma 64), we obtain:

(12.21) $\Gamma, \Gamma''; p \vdash \begin{array}{l} \{F' * H''[\sigma'] * E'[\sigma'] * p \neq \texttt{null}\} \\ c'' : U'[\sigma'] \\ \{(\texttt{ex}\, U'[\sigma']\ \alpha'')(F' * H''[\sigma'] * H'[\sigma'])\} \end{array}$

We weaken (12.8) by extending the type environment to (Γ, Γ''):

(12.22) $\Gamma, \Gamma''; r \vdash \{F' * (\texttt{ex}\, U[\sigma']\ \alpha'')(\alpha'' == \ell * H[\sigma'])\} c' : \texttt{void}\{G\}$

Using the natural deduction rules, one can show the following;

(12.23) $\begin{array}{lll} \Gamma, \Gamma''; r; & & (\texttt{ex}\, U'[\sigma']\ \alpha'')(F' * H''[\sigma'] * H'[\sigma']) \\ & \vdash & F' * (\texttt{ex}\, U[\sigma']\ \alpha'')(\alpha'' == \ell * H[\sigma']) \end{array}$

Thus, by logical consequence (Lemma 62), we get:

(12.24) $\Gamma, \Gamma''; r \vdash \{(\texttt{ex}\, U'[\sigma']\ \alpha'')(F' * H''[\sigma'] * H'[\sigma'])\} c' : \texttt{void}\{G\}$

Now we can apply the derived rule for "bind" (Lemma 65) to (12.21) and (12.24):

(12.25) $\Gamma, \Gamma''; p \vdash \{F' * H''[\sigma'] * E'[\sigma'] * p \neq \texttt{null}\} \ell \leftarrow c''; c' : \texttt{void}\{G\}$

Because (12.19), (12.18) and (12.25) hold, we can apply (Thread) to obtain:

(12.26) $\mathscr{R} \vdash o \text{ is } (s \text{ in } \ell \leftarrow c''; c') : \diamond$

The rest is routine.

Case 13, (Red Return):

$$\langle h, ts \mid o \text{ is } (s \text{ in } \ell = \texttt{return}(v); c') \rangle \rightarrow \langle h, ts \mid o \text{ is } (s \text{ in } \ell = v; c') \rangle$$

In this case, we can further instantiate c and st' as follows:

(13.1) $c = \texttt{return}\, v; c'$
(13.2) $st' = \langle h, ts \mid o \text{ is } (s \text{ in } \ell = v; c') \rangle$

The last rule in (12)'s derivation is (Return). Its premises are:

(13.3) $\Gamma = (\Gamma'', \ell : U)$
(13.4) $\Gamma'', \Gamma' \vdash v : T$
(13.5) $\Gamma'', \Gamma'; o; F \vdash H[v/\alpha]$
(13.6) $T <: U$
(13.7) $\Gamma, \Gamma'; o \vdash \{(\texttt{ex}\, T\, \alpha)(\alpha == \ell * H)\} c' : \texttt{void}\{G\}$

From (11) and (13.5) we obtain:

(13.8) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models H[v/\alpha] : \checkmark$

By (Var Set) we obtain the following statement. (The rule premise $\ell \notin H[v/\alpha]$ follows from (13.5) and $\ell \notin \text{dom}(\Gamma'', \Gamma')$.)

(13.9) $\Gamma,\Gamma';o \vdash \{H[v/\alpha]\}\ell\texttt{=}v\{H[v/\alpha]*\ell\texttt{==}v\}$

By natural deduction, $(\Gamma,\Gamma';o;H[v/\alpha]*\ell\texttt{==}v \vdash \texttt{(ex}\ T\ \alpha\texttt{)(}\alpha\texttt{==}\ell*H\texttt{))}$. We apply Lemma 62 and (Seq) to (13.7) and (13.9):

(13.10) $\Gamma,\Gamma';o \vdash \{H[v/\alpha]\}\ell\texttt{=}v;\ c' : \texttt{void}\{G\}$

Then we apply (Thread) to (13.8) and (13.10):

(13.11) $\mathscr{R} \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}v;\ c') : \diamond$

The rest is routine.

Case 14, (Red Fork):

$$\frac{h(p)_1 = C\texttt{<}\overline{\pi}\texttt{>} \quad p \notin \mathsf{dom}(ts),\{o\} \quad \mathsf{mbody}(\texttt{run},C\texttt{<}\overline{\pi}\texttt{>}) = \texttt{<>(this)}.c_r \quad c'' = c_r[p/\texttt{this}]}{\langle h,\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}p\texttt{.fork();}\ c')\rangle \rightarrow \langle h,\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=null;}\ c') \mid p\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c'')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(14.1) $c = \ell\texttt{=}p\texttt{.fork();}\ c'$
(14.2) $st' = \langle h,\ ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=null;}\ c') \mid p\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c'')\rangle$

The last rules in (12)'s derivation are (Seq) preceded by (Call). From the rule premises, we obtain:

(14.3) $\Gamma,\Gamma' \vdash p : t\texttt{<}\overline{\pi}'\texttt{>}$
(14.4) $\mathsf{mtype}(\texttt{fork},C'\texttt{<}\overline{\pi}'\texttt{>}) =$
$\texttt{final req}\ G'_{req}\texttt{;ens (ex void}\ \alpha'\texttt{)(true); void fork(}C'\texttt{<}\overline{\pi}'\texttt{>}\ \texttt{this)}$
(14.5) $\sigma' = (p/\texttt{this})$
(14.6) $\Gamma,\Gamma';r;F \vdash F'*F''*\ G'_{req}[\sigma']$
(14.7) $\texttt{void} <: \Gamma(\ell)$
(14.8) $\Gamma,\Gamma';r \vdash \{F'*\texttt{(ex void}\ \alpha'\texttt{)(}\alpha'\texttt{==}\ell*\ \mathsf{all}\cdot\texttt{true)}\}c' : \texttt{void}\{G\}$
(14.9) $\ell \notin F'$

We know that, by definition, `fork`'s precondition is equal to `run`'s precondition:

(14.10) $\mathsf{mtype}(\texttt{run},C'\texttt{<}\overline{\pi}'\texttt{>}) =$
$\texttt{req}\ G'_{req}\texttt{;ens (ex void}\ \alpha'\texttt{)(}G'_{ens}\texttt{); void run(}C'\texttt{<}\overline{\pi}'\texttt{>}\ \texttt{this)}$

By the same argumentation as in the proof case (Red Call), we can show the following:

(14.11) $C\texttt{<}\overline{\pi}\texttt{>} = h(p)_1 = \Gamma(p) <: C'\texttt{<}\overline{\pi}'\texttt{>}$
(14.12) $\Gamma_{\mathsf{hp}} \vdash \mathsf{mtype}(C\texttt{<}\overline{\pi}\texttt{>},\texttt{run}) <: \mathsf{mtype}(C'\texttt{<}\overline{\pi}'\texttt{>},\texttt{run})$

Therefore, these two method types are related in the following way:

(14.13) $\mathsf{mtype}(\texttt{run},C\texttt{<}\overline{\pi}\texttt{>}) =$
$\texttt{req}\ G_{req}\texttt{;ens (ex void}\ \alpha'\texttt{)(}G_{ens}\texttt{); void run(}C\texttt{<}\overline{\pi}\texttt{>}\ \texttt{this)}$
(14.14) $\texttt{this} : C\texttt{<}\overline{\pi}\texttt{>;this;true} \vdash \texttt{(}G'_{req}\ \texttt{-*}\ G_{req}\texttt{)*(fa void}\ \overline{\alpha}'\texttt{)(}G_{ens}\ \texttt{-*}\ G'_{ens}\texttt{)}$

Let $C\texttt{<}\overline{T}\ \overline{\alpha}\texttt{>} \in ct$ and $\Gamma'' = (\overline{\alpha} : \overline{T}, \texttt{this} : C\texttt{<}\overline{\alpha}\texttt{>})$. From the premises of rule (Mth) for C.`run`, we obtain:

(14.15) $\Gamma'';\texttt{this} \vdash \{G_{req}\texttt{*this !=null}\}c_r : \texttt{void}\{\texttt{(ex void}\ \alpha'\texttt{)(}G_{ens}\texttt{)}\}$

Because $(\Gamma_{\mathsf{hp}} \vdash h : \diamond)$, we know that $(\Gamma_{\mathsf{hp}} \vdash \overline{\pi} : \overline{T}[\overline{\pi}/\overline{\alpha}])$. By applying substitutivity (Lemmas 49 and 61) to (14.14) and (14.15), we obtain:

(14.16) $\Gamma_{\mathsf{hp}}; p; \texttt{true} \vdash (G'_{req}[\sigma'] \mathrel{-\!\!*} (G_{req}[\sigma']) * (\texttt{fa void}\,\alpha')(G_{ens}[\sigma'] \mathrel{-\!\!*} G'_{ens}[\sigma'])$
(14.17) $\Gamma_{\mathsf{hp}}; p \vdash \{G_{req}[\sigma'] * p\,\texttt{!=null}\}c'' : \texttt{void}\{(\texttt{ex void}\,\alpha')(G_{ens}[\sigma'])\}$

From (11) and (14.6), we get $(\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] * G'_{req}[\sigma'] : \checkmark)$. Applying (14.16) to this (recall that $\Gamma_{\mathsf{hp}}[\sigma] = \Gamma_{\mathsf{hp}}$), we obtain:

(14.18) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] * G_{req}[\sigma'] : \checkmark$

Then, by definition of semantic validity, there exist $\mathscr{R}_o$ and $\mathscr{R}_p$ such that:

(14.19) $\mathscr{R}_o * \mathscr{R}_p = \mathscr{R}$
(14.20) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}_o; s \models F' : \checkmark$
(14.21) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}_p; s \models G_{req}[\sigma'] * p\,\texttt{!=null} : \checkmark$

From (14.21) and (14.17), it follows that:

(14.22) $\mathscr{R}_p \vdash p\ \mathsf{is}\ (\emptyset\ \mathsf{in}\ c'') : \diamond$

Applying admissibility of logical consequence (Lemma 62) to (14.8), we get:

(14.23) $\Gamma, \Gamma'; r \vdash \{F' * (\texttt{ex void}\ \alpha')(\alpha' == \ell * \texttt{true}) * \ell == \texttt{null}\}c' : \texttt{void}\{G\}$

Then by (Var Set):

(14.24) $\Gamma, \Gamma'; r \vdash \{F' * (\texttt{ex void}\ \alpha')(\alpha' == \ell * \texttt{true})\}\ell\texttt{=null}; c' : \texttt{void}\{G\}$

The existential formula can be validated by instantiating α' by $s(\ell)$. Therefore, (14.20) gives us:

(14.25) $\Gamma, \Gamma' \vdash \mathscr{E}; \mathscr{R}_o; s \models F' * (\texttt{ex void}\ \alpha')(\alpha' == \ell * \texttt{true}) : \checkmark$

From (14.25) and (14.24) it follows that:

(14.26) $\mathscr{R}_o \vdash o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=null}; c') : \diamond$

The rest is routine.

Case 15, (Red Join):

$\langle h, ts' \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=}p\texttt{.join();}\ c') \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ v)\rangle \rightarrow \langle h, ts' \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=null;}\ c') \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ v)\rangle$

In this case, we can further instantiate c and st' as follows:

(15.1) $ts = ts' \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ v)$
(15.2) $c = \ell\texttt{=}p\texttt{.join();}\ c'$
(15.3) $st' = \langle h, ts' \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \ell\texttt{=null;}\ c') \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ v)\rangle$

The last rules in (12)'s derivation are (Seq) preceded by (Call). From the rule premises we obtain:

(15.4) $\Gamma, \Gamma' \vdash p : C'\texttt{<}\overline{\pi}'\texttt{>}$
(15.5) $\mathsf{mtype}(\texttt{join}, C'\texttt{<}\overline{\pi}'\texttt{>}) =$
$\texttt{final req true; ens (ex void}\,\alpha')(G'_{ens})\texttt{; void join(}C'\texttt{<}\overline{\pi}'\texttt{> this)}$
(15.6) $\sigma' = (p/\texttt{this})$
(15.7) $\Gamma, \Gamma'; r; F \vdash F' * fr' \cdot \texttt{Perm}(p\texttt{[join]}, 1) * G'_{req}[\sigma']$
(15.8) $\texttt{void} <: \Gamma(\ell)$
(15.9) $\Gamma, \Gamma'; r \vdash \{F' * (\texttt{ex void}\ \alpha')(\overline{\alpha}' == \ell * fr' \cdot G'_{ens}[\sigma'])\}c' : \texttt{void}\{G\}$
(15.10) $\ell \notin F'$

We know that `join`'s postcondition, by definition, is equal to `run`'s postcondition:

(15.11) $\mathsf{mtype}(\texttt{run}, C\texttt{<}\bar{\pi}'\texttt{>}) =$
$\texttt{req}\ G''_{req}\texttt{;ens (ex void}\ \alpha'\texttt{)(}G'_{ens}\texttt{); void run(}C\texttt{<}\bar{\pi}'\texttt{>}\ \texttt{this)}$

Let $h(p)_1 = C\texttt{<}\bar{\pi}\texttt{>}$. By the same argumentation as in the proof case (Red Call), we can show the following:

(15.12) $C\texttt{<}\bar{\pi}\texttt{>} = h(p)_1 = \Gamma(p) <: C'\texttt{<}\bar{\pi}'\texttt{>}$
(15.13) $\Gamma_{\mathsf{hp}} \vdash \mathsf{mtype}(\texttt{run}, C\texttt{<}\bar{\pi}\texttt{>}) <: \mathsf{mtype}(\texttt{run}, C'\texttt{<}\bar{\pi}'\texttt{>})$

Therefore, these two method types are related in the following way:

(15.14) $\mathsf{mtype}(\texttt{run}, C\texttt{<}\bar{\pi}\texttt{>}) =$
$\texttt{req}\ G_{req}\texttt{;ens (ex void}\ \alpha'\texttt{)(}G_{ens}\texttt{); void run(}C\texttt{<}\bar{\pi}\texttt{>}\ \texttt{this)}$
(15.15) $\texttt{this} : C\texttt{<}\bar{\pi}\texttt{>}\texttt{;this;true} \vdash (G''_{req} \mathrel{-\!\!*} G_{req}) * \texttt{(fa void}\ \alpha'\texttt{)}(G_{ens} \mathrel{-\!\!*} G'_{ens})$

From (11) and (15.7) we obtain:

(15.16) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}; s \models F'[\sigma] * fr' \cdot \texttt{Perm}(p\texttt{[join]}, 1) : \checkmark$

By the semantics of $*$, there exist $\mathscr{R}^{o,1}$ and $\mathscr{R}^{o,2}$ such that:

(15.17) $\mathscr{R} = \mathscr{R}^{o,1} * \mathscr{R}^{o,2}$
(15.18) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{o,1}; s \models F'[\sigma] : \checkmark$
(15.19) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{o,2}; s \models fr' \cdot \texttt{Perm}(p\texttt{[join]}, 1) : \checkmark$

The last of these statements means that:

(15.20) $[\![fr']\!] \leq \mathscr{R}^{o,2}_{\mathsf{loc}}(p, \texttt{join})$

Recall that $(\mathscr{R}_{ts} \vdash ts : \diamond)$, by (7), and $ts = ts' \mid p\ \mathsf{is}\ (s'\ \mathsf{in}\ v)$, by (15.1). The premises of the last rule of $(\mathscr{R}_{ts} \vdash ts : \diamond)$'s derivation are:

(15.21) $\mathscr{R}^{ts} = \mathscr{R}^{ts'} * \mathscr{R}^{p}$
(15.22) $\mathscr{R}^{ts'} \vdash ts' : \diamond$
(15.23) $\mathscr{R}^{p} \vdash p\ \mathsf{is}\ (s'\ \mathsf{in}\ v) : \diamond$

Let $\mathscr{Q} = \mathscr{R}^{p}_{\mathsf{glo}}$. Recall that the $*$-composition of two resources is only defined if they both have the same global permission table. So $\mathscr{Q} = \mathscr{R}^{ts}_{\mathsf{glo}} = \mathscr{R}_{\mathsf{glo}} = \mathscr{R}^{o}_{\mathsf{glo}}$ hold, too. From $(\mathscr{R}^{p} \vdash p\ \mathsf{is}\ (s'\ \mathsf{in}\ v) : \diamond)$ we obtain:

(15.24) $\Gamma''[\sigma'''] \vdash \mathscr{E}; \mathscr{R}^{p}; s' \models fr_p \cdot G_{ens}[\sigma'][v/\alpha'] : \checkmark$
(15.25) $\mathscr{Q}(p, \texttt{join}) \leq [\![fr_p]\!]$

We know that $\Gamma''_{\mathsf{hp}}[\sigma'''] = \Gamma''_{\mathsf{hp}} = h(o)_1 = \Gamma_{\mathsf{hp}} = \Gamma_{\mathsf{hp}}[\sigma]$. Because $G_{ens}[\sigma']$ does not contain free variables. we can restrict the stack in (15.24):

(15.26) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{p}; \emptyset \models fr_p \cdot G_{ens}[\sigma'][v/\alpha'] : \checkmark$

We define: $\sigma'' = (\sigma', v/\alpha')$. We have that $[\![fr']\!] \leq \mathscr{R}^{o,2}_{\mathsf{loc}}(p, \texttt{join}) \leq \mathscr{Q}(p, \texttt{join}) \leq [\![fr_p]\!]$. Therefore $fr_p - fr'$ exists (by Lemma 88). By distributivity (Lemma 92), we have $fr_p \cdot G_{ens}[\sigma''] = ((fr_p - fr') + fr') \cdot G_{ens}[\sigma''] \mathrel{*\!\!-\!\!*} (fr_p - fr') \cdot G_{ens}[\sigma''] * fr' \cdot G_{ens}[\sigma'']$. Therefore, (15.26) implies the following statement:

(15.27) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{p}; \emptyset \models (fr_p - fr') \cdot G_{ens}[\sigma''] * fr' \cdot G_{ens}[\sigma'']$.

By definition of semantic validity, there exist $\mathscr{R}^{p,1}$ and $\mathscr{R}^{p,2}$ such that:

(15.28) $\mathscr{R}^{p} = \mathscr{R}^{p,1} * \mathscr{R}^{p,2}$

(15.29) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{p,1}; \emptyset \models (fr_p - fr') \cdot G_{ens}[\sigma''] : \checkmark$
(15.30) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{p,2}; \emptyset \models fr' \cdot G_{ens}[\sigma''] : \checkmark$

Applying (15.15), Lemma 86 and assumption (17) to (15.30), we get:

(15.31) $\Gamma[\sigma] \vdash \mathscr{E}; \mathscr{R}^{p,2}; \emptyset \models fr' \cdot G'_{ens}[\sigma''] : \checkmark$

We now define:

(15.32) $\mathscr{Q}' \triangleq \mathscr{Q}[(p, \texttt{join}) \mapsto (\mathscr{Q}(p, \texttt{join}) - [\![fr']\!])]$
(15.33) $(\mathscr{R}^{ts'})' \triangleq (\mathscr{R}^{ts'}_{\mathsf{hp}}, \mathscr{R}^{ts'}_{\mathsf{loc}}, \mathscr{Q}')$
(15.34) $(\mathscr{R}^{o,1})' \triangleq (\mathscr{R}^{o,1}_{\mathsf{hp}}, \mathscr{R}^{o,1}_{\mathsf{loc}}, \mathscr{Q}')$
(15.35) $(\mathscr{R}^{p,2})' \triangleq (\mathscr{R}^{p,2}_{\mathsf{hp}}, \mathscr{R}^{p,2}_{\mathsf{loc}}, \mathscr{Q}')$
(15.36) $(\mathscr{R}^{p})' \triangleq (\mathscr{R}^{p,1}_{\mathsf{hp}}, \mathscr{R}^{p,1}_{\mathsf{loc}}, \mathscr{Q}')$
(15.37) $(\mathscr{R}^{o})' \triangleq (\mathscr{R}^{o,1})' * (\mathscr{R}^{p,2})'$

It now suffices to show the following claims:

(15.38) $(\mathscr{R}^{ts'})' \vdash ts' : \diamond$ goal
(15.39) $(\mathscr{R}^{p})' \vdash p \text{ is } (s' \text{ in } v) : \diamond$ goal
(15.40) $(\mathscr{R}^{o})' \vdash o \text{ is } (s \text{ in } \ell\texttt{=null}; c') : \diamond$ goal

Goal (15.38) is a consequence of $(\mathscr{R}^{ts'} \vdash ts' : \diamond)$ and Lemma 78. To show goal (15.39) we use (15.29) and Lemma 78. To reestablish assumption (17) for goal (15.39), we use the restriction that `run`'s postcondition does not mention `this[join]` (imposed by rule (Mth Type)). So we are left with goal (15.40): Applying Lemma 78 to (15.18) and (15.31), we obtain:

(15.41) $\Gamma[\sigma] \vdash \mathscr{E}; (\mathscr{R}^{o})'; s \models F'[\sigma] * fr' \cdot G'_{ens}[\sigma''] : \checkmark$

Because $\sigma'' = (\sigma', v/\alpha')$ and furthermore because all values of type `void` are equal to `null`, we obtain:

(15.42) $\Gamma[\sigma] \vdash \mathscr{E}; (\mathscr{R}^{o})'; s \models F'[\sigma] * \texttt{(ex void } \alpha')\texttt{(}\alpha' \texttt{==} \ell * fr' \cdot G'_{ens}[\sigma']\texttt{)} : \checkmark$

On the other hand, applying admissibility of logical consequence (Lemma 62) and (Var Set) to (15.9) (like at the end of proof case (Red Fork)), we get:

(15.43) $\Gamma, \Gamma'; r \vdash \{F' * \texttt{(ex void } \alpha')\texttt{(}\alpha' \texttt{==} \ell * fr' \cdot G'_{ens}[\sigma']\texttt{)}\}\ell\texttt{=null; } c' : \texttt{void}\{G\}$

Our goal (15.40), now follows from (15.42) and (15.43).

Case 16, (Red Assert):

$$\frac{}{\langle h, ts \mid o \text{ is } (s \text{ in } \texttt{assert(}H\texttt{); } c')\rangle \rightarrow \langle h, ts \mid o \text{ is } (s \text{ in } c')\rangle}$$

In this case, we can further instantiate c and st' as follows:

(16.1) $c = \texttt{assert(}H\texttt{); } c'$
(16.2) $st' = \langle h, ts \mid o \text{ is } (s \text{ in } c')\rangle$

The last rules in (12)'s derivation are (Seq) preceded by (Assert). From the rule premises, we get:

(16.3) $\Gamma[\sigma]; r; F \vdash F'$
(16.4) $\Gamma, \Gamma'; r; F' \vdash H$

(16.5) $\Gamma,\Gamma';r \vdash \{F'\}c' : \texttt{void}\{G\}$

From (11) and (16.3) we obtain:

(16.6) $\Gamma[\sigma] \vdash \mathscr{E};\mathscr{R};s \models F' : \checkmark$.

We apply (Thread) to (16.6) and (16.5). We obtain $(\mathscr{R} \vdash o \text{ is } (s \text{ in } c') : \diamond)$. We then apply (Cons Pool) and (State) to obtain $st' : \diamond$. □

V Data Race Freedom, Null Error Freeness, Partial Correctness

After all this hard work, we can now easily prove several corollaries of the preservation theorem. Let main be a distinguished object id. We define the *initial state*:

$$\text{init}(c) \triangleq \langle \{\text{main} \mapsto (\texttt{Thread}, \emptyset)\}, \text{main is } (\emptyset \text{ in } c)\rangle$$

Lemma 93 *If* $(ct,c) : \diamond$, *then* $\text{init}(c) : \diamond$.

Proof. Suppose $(ct,c) : \diamond$. By definition, this means $ct : \diamond$ and $\text{main} : \texttt{Thread};\text{main} \vdash \{\texttt{true}\}c : \texttt{void}\{\texttt{true}\}$. Let $h = \{\text{main} \mapsto (\texttt{Thread}, \emptyset)\}$. Let $\mathscr{R} = (h, \mathbf{0}, \mathbf{1})$. Let $\mathscr{E}$ be some predicate environment such that $\mathscr{F}_{ct}(\mathscr{E}) = \mathscr{E}$ (which exists by Theorem 8). Then $(\text{main} : \texttt{Thread} \vdash \mathscr{E};\mathscr{R};\emptyset \models \texttt{true} : \checkmark)$. Now it is easy to check that the premises of (Thread) are satisfied (pick $\sigma = \Gamma' = \emptyset$). It follows that $(\mathscr{R} \vdash \text{main is } (\emptyset \text{ in } c) : \diamond)$. Thus, $\text{init}(c) : \diamond$, by (State). □

A pair (hc, hc') of head commands is called a *data race* iff $hc = (\mathit{fin}\ o.f = v)$ and either $hc' = (\mathit{fin}'\ o.f = v')$ or $hc' = (\ell = o.f)$ for some $o, f, v, v', \ell, \mathit{fin}, \mathit{fin}'$.

Proof of Theorem 1 (Verified Programs are Data Race Free). *If* $(ct,c) : \diamond$ *and* $\text{init}(c) \rightarrow_{ct}^{*} \langle h, ts \mid o_1 \text{ is } (s_1 \text{ in } hc_1;c_1) \mid o_2 \text{ is } (s_2 \text{ in } hc_2;c_2)\rangle$, *then* (hc_1, hc_2) *is not a data race.*

Proof. Let $(ct,c) : \diamond$, $st = \langle h, ts \mid o_1 \text{ is } (s_1 \text{ in } hc_1;c_1) \mid o_2 \text{ is } (s_2 \text{ in } hc_2;c_2)\rangle$, and $\text{init}(c) \rightarrow_{ct}^{*} st$. By $\text{init}(c) : \diamond$ (Lemma 93) and preservation (Theorem 5), we know that $st : \diamond$. Suppose, towards a contradiction, that (hc_1, hc_2) is a data race. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be resources $\mathscr{R}$, $\mathscr{R}'$ and a heap cell $o.f$ such that $\mathscr{R} \vdash o_1 \text{ is } (s_1 \text{ in } hc_1;c_1) : \diamond$, $\mathscr{R}' \vdash o_2 \text{ is } (s_2 \text{ in } hc_2;c_2) : \diamond$, $\mathscr{R}\#\mathscr{R}'$, $\mathscr{R}_{\text{loc}}(o,f) = 1$ and $\mathscr{R}'_{\text{loc}}(o,f) > 0$. But then $\mathscr{R}_{\text{loc}}(o,f) + \mathscr{R}'_{\text{loc}}(o,f) > 1$, in contradiction to $\mathscr{R}\#\mathscr{R}'$. □

A head command hc is called a *null error* iff $hc = (\ell = \texttt{null}.f)$ or $hc = (\mathit{fin}\ \texttt{null}.f = v)$ or $hc = (\ell = \texttt{null}.m\texttt{<}\overline{\pi}\texttt{>}(\overline{v}))$ for some $\ell, \mathit{fin}, f, v, m, \overline{\pi}, \overline{v}$.

Proof of Theorem 2 (Verified Programs are Null Error Free). *If* $(ct,c) : \diamond$ *and* $\text{init}(c) \rightarrow_{ct}^{*} \langle h, ts \mid o \text{ is } (s \text{ in } hc;c)\rangle$, *then* hc *is not a null error.*

Proof. Let $(ct,c) : \diamond$, $st = \langle h, ts \mid o \text{ is } (s \text{ in } hc;c)\rangle$, and $\text{init}(c) \rightarrow_{ct}^{*} st$. By $\text{init}(c) : \diamond$ (Lemma 93) and preservation (Theorem 5), we know that $st : \diamond$. Suppose, towards a contradiction, that hc is a null error. Then $hc = (\ell = \texttt{null}.f)$ or $hc = (\mathit{fin}\ \texttt{null}.f = v)$ or $hc = (\ell = \texttt{null}.m\texttt{<}\overline{\pi}\texttt{>}(\overline{v}))$.

Suppose first that $hc = (\ell = \texttt{null}.f)$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathscr{E}$, $\mathscr{R}$, s, π, u such that either $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$

`PointsTo(null[`f`],`π`,`u`)` or $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `Pure(null.`f`)`. But neither of these statements hold, by definition of $\models$.

Suppose now that $hc = (\mathit{fin}\ \texttt{null}.f = v)$ An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathscr{E}$, $\mathscr{R}$, s, T such that $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `PointsTo(null[`f`],1,`T`)`. But this is false, by definition of $\models$.

Suppose finally that $hc = (\ell$`=null.`m`<`$\overline{\pi}$`>(`$\overline{v}$`)`$)$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathscr{E}$, $\mathscr{R}$, s such that $\Gamma \vdash \mathscr{E};\mathscr{R};s \models$ `null !=null`, which is obviously false. □

Proof of Theorem 3 (Partial Correctness).
If $(ct,c) : \diamond$ *and* $\mathsf{init}(c) \rightarrow^*_{ct} \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{assert(}F\texttt{);}\ c)\rangle$, *then* $(\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{Q});s \models F[\sigma])$ *for some* $\Gamma, \mathscr{E} = \mathscr{F}_{ct}(\mathscr{E}), \mathscr{P}, \mathscr{Q}$ *and* $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$.

Proof. Let $(ct,c) : \diamond$, $st = \langle h, ts \mid o\ \mathsf{is}\ (s\ \mathsf{in}\ \texttt{assert(}F\texttt{);}\ c)\rangle$, and $\mathsf{init}(c) \rightarrow^*_{ct} st$. By $\mathsf{init}(c) : \diamond$ (Lemma 93) and preservation (Theorem 5), we know that $st : \diamond$. An inspection of the last rules of $(st : \diamond)$'s derivation reveals that there must then be Γ, $\mathscr{E} = \mathscr{F}_{ct}(\mathscr{E})$, $\mathscr{R}$, $\sigma \in \mathsf{LogVar} \rightharpoonup \mathsf{SpecVal}$ such that $(\Gamma \vdash \mathscr{E};(h,\mathscr{P},\mathscr{Q});s \models F[\sigma])$. □

We could strengthen the partial correctness theorem and universally quantify over $\mathscr{E}$ up front, if our judgment for good states took the predicate environment $\mathscr{E}$ as an argument ("$\mathscr{E} \vdash st : \diamond$").

References

[1] M. Abadi, C. Flanagan, S. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2), 2006.

[2] A. W. Appel, S. Blazy. Separation logic for small-step Cminor. In *TPHOL*, 2007.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6), 2004.

[4] J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[5] K. Bierhoff, J. Aldrich. Modular typestate verification of aliased objects. In *OOPSLA*, 2007.

[6] R. Bornat, P. O'Hearn, C. Calcagno, M. Parkinson. Permission accounting in separation logic. In *POPL*, New York, NY, USA, 2005. ACM Press.

[7] C. Boyapati, R. Lee, M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.

[8] J. Boyland. Checking interference with fractional permissions. In R. Cousot, ed., *SAS*, vol. 2694 of *LNCS*. Springer-Verlag, 2003.

[9] J. Boyland, W. Retert. Connecting effects and uniqueness with adoption. In *POPL*, 2005.

[10] S. Brookes. A semantics for concurrent separation logic. In *Conference on Concurrency Theory*, vol. 3170 of *LNCS*. Springer-Verlag, 2004.

[11] S. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. *ENTCS*, 158, 2006.

[12] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, vol. 80 of *ENTCS*. Elsevier, 2003.

[13] R. DeLine, M. Fähndrich. Typestates for objects. In *ECOOP*, 2004.
[14] D. Distefano, P. W. O'Hearn, H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
[15] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
[16] A. Igarashi, B. Pierce, P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.
[17] S. Ishtiaq, P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
[18] B. Jacobs, J. Smans, F. Piessens, W. Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *ICFEM*, 2006.
[19] G. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS*, 2006.
[20] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (Second Edition)*. Addison-Wesley, Boston, MA, USA, 1999.
[21] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, J. Kiniry. *JML Reference Manual*, 2005. In Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.
[22] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, 1998.
[23] J. Manson, W. Pugh, S. V. Adve. The Java memory model. In *POPL*, 2005.
[24] P. O'Hearn. Resources, concurrency and local reasoning. *TCS*, 375(1–3), 2007.
[25] P. W. O'Hearn, D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.
[26] M. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.
[27] M. Parkinson, G. Bierman. Separation logic and abstraction. In *POPL*, 2005.
[28] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet. Enforcing high-level security properties for applets. In *CARDIS 2004*, 2004.
[29] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Press.
[30] J. C. Reynolds. Towards a grainless semantics for shared variable concurrency. In K. Lodaya, M. Mahajan, eds., *FSTTCS*, vol. 3328 of *LNCS*. Springer-Verlag, 2004.
[31] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, 1993.
[32] H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *SPACE*, 2001.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803